APPENDIX A

Cellular Basestation Modem Engine (CBME)
Virtual Machine Interface Specification

Cellular Basestation Modem Engine (CBME) Virtual Machine Interface Specification

Preliminary Specification

Document Version 2.01

Morphics Technology, Inc.
675 Campbell Technology Parkway, Suite 200
Campbell, CA 95008, USA
Tel: 408.369.7227
Fax: 408.369.7210

Table of Contents

Table of Tables

Table of Figures

1 Introduction

This document describes the software library used to control the Morphics CBME (Cellular Basestation Modem Engine). The software library is called the VMI ((Virtual Machine Interface). The VMI is in ANSI 'C' source code format, and will need to be compiled, linked, and integrated into an application development environment. Benefits of the VMI library include:

- Decoupled from CBME ASIC implementation
- Full access to all CBME resources
- Customer ease of use
- ANSI C compatible
- Generic RTOS compatibility

2 VMI Theory of Operation

The CBME VMI is divided into several primary objects, each with method functions. The combination of these objects and their method functions allow a user to access the full functionality and capability of the CBME, and at the same time, keeps the interface independent of the underlying ASIC architecture.

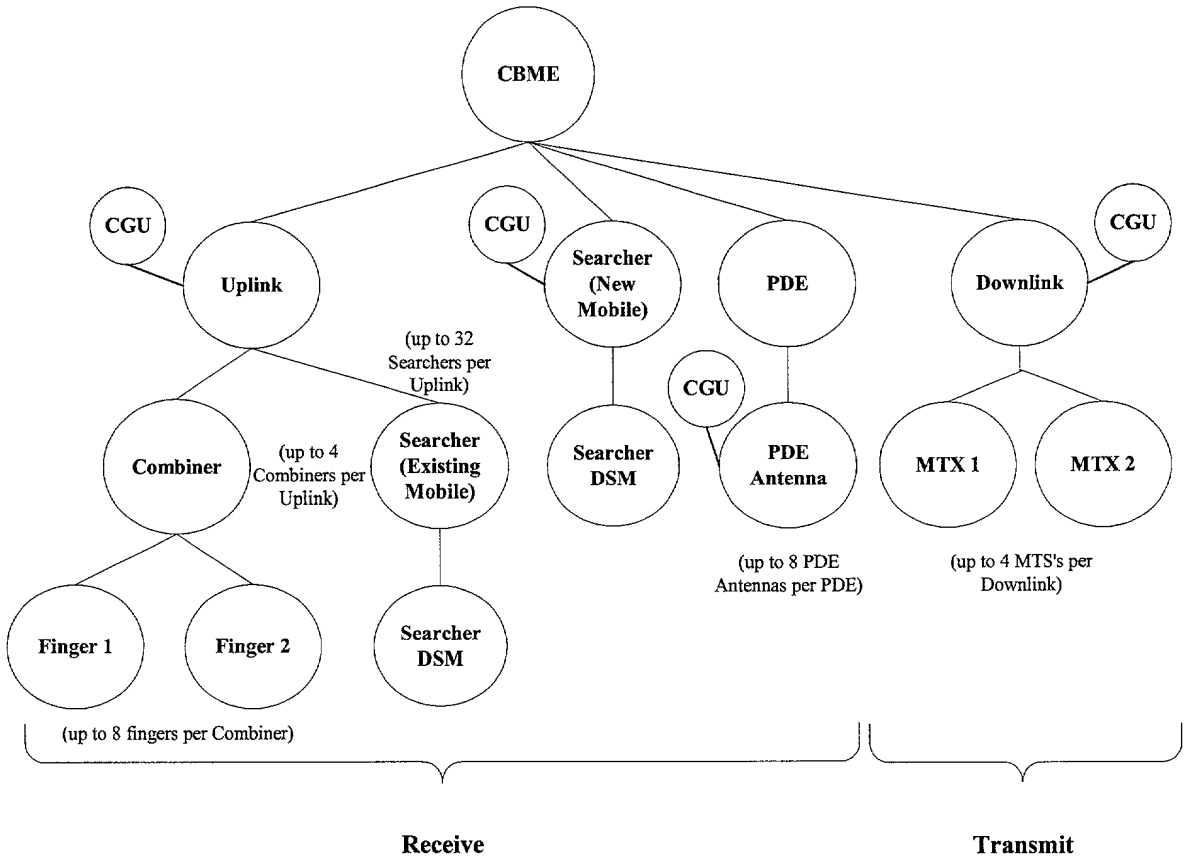
Table 2-1: CBME Objects

Objects
CBME
CGU
Searcher
Searcher DSM
Preamble Detection Engine
Preamble Detection Engine Antenna
Finger
Combiner
Uplink
Downlink
MTX

A detailed description of each object, its relationship to other objects, and where it fits in the hierarchy is now discussed. All VMI functions and data types are included via the **cbme.h** file.

Figure 2-1 below shows a sample hierarchy of CBME objects for a single CBME ASIC.

Figure 2-1: Example CBME Object Hierarchy



A number of object relationship rules are implied in Figure 2-1 above. These rules are detailed in Table 2-2.

Table 2-2 : Object Relationship Rules

Object	Description	Rule(s)	Maximum Objects Supported With Current Revision of ASIC
CBME	Cellular Basestation Modem Engine. This is the highest-level object. All other objects for the same CBME ASIC are associated, directly or indirectly, with this object.	All other objects, for a specific CBME ASIC, are derived from the CBME object.	One CBME object per physical CBME ASIC.
CGU	Code Generation Unit.	Required for Uplinks, 'new mobile' Searchers, Downlinks, and Preamble Detection Engine Antennas.	8 'base-line' CGU's can be customized into an unlimited number of CGU objects.
Searcher	Searches for mobile channels. Each Searcher has one DSM associated with it. A Searcher is configured to either search for 'new mobiles' or 'existing mobiles'.	Each searcher is associated with one Searcher DSM.	Determined by the chipping rate.
Searcher DSM (Dwell State Machine)	Used to configure the Searcher algorithms.	Each Searcher DSM is associated with one or more searchers.	16 total dwell states can be used as single dwells or as multi-dwells. 4 multi-dwells allowed.
Finger	Fingers are used to track mobiles that the searcher has acquired.	Fingers must be combined using the combiner object.	Determined by chipping rate.
Combiner	Combines one or more fingers and sums them up via a combining rule.	A combiner is associated with one to 16 fingers.	Determined by chipping rate.
Uplink	This object associates combiners and 'existing mobile' searchers that are resourced to a single mobile uplink.	All combiners and 'existing mobile' searchers must be added to an Uplink.	Uplinks group other objects, and as such, consume no CBME resources. There is no limit to the number of Uplinks that can be declared.

Object	Description	Rule(s)	Maximum Objects Supported With Current Revision of ASIC
Preamble Detection Engine (PDE)	A faster version of a 'new mobile' searcher, but with less configurability.	Must have at least one Preamble Detection Engine Antenna attached. Maximum of 8 antennas can be attached.	32
Preamble Detection Engine Antenna	Antenna for a Preamble Detection Engine.	Must be attached to a Preamble Detection Engine.	24
Downlink	Primary Physical Transmitter channel. Represents one primary channel.	None.	TBD
MTX	Multicode Transmitter Object. Represents one multicode channel.	All MTX objects must be added to a Downlink object.	TBD

2.1 *Memory Usage*

The user has complete control over memory management. All VMI objects are either declared at compile time or can be dynamically allocated at run time. The VMI library does not perform any internal dynamic memory allocation. A few short examples follow on how the VMI objects can be statically or dynamically allocated.

It is presumed that all tasks interfacing to the VMI will reside in a single memory space.

2.1.1 Static Allocation Example of VMI Objects

```
#include "cbme.h"

CBME     my_cbme;        /* declare a CBME object      */
SEARCHER my_searcher;    /* declare a Searcher object */

/* create a new CBME object */
error_code = CBME_New(&my_cbme, (UINT32 *)0x12345678, M_SEPARATE_INT);
      .
      .
/* create a new Searcher object */
error_code = Searcher_New(&my_cbme,
                          &my_searcher,
              .           M_EXISTING_MOBILE_SEARCHER);
```

2.1.2 Dynamic Allocation Example of VMI Objects

```
#include "cbme.h"

CBME      *p_cbme;       /* pointer to CBME object      */
SEARCHER  *p_searcher;   /* pointer to Searcher object  */

/* allocate memory for CBME object */
p_cbme = (CBME *)(malloc (sizeof(CBME)));

/* create a new CBME */
error_code = CBME_New(p_cbme, (UINT32 *)0x12345678, M_SEPARATE_INT);
      .
      .
/* allocate memory for a Searcher object */
p_searcher = (SEARCHER *)(malloc (sizeof(SEARCHER)));

/* create a new Searcher object */
error_code = Searcher_New(p_cbme,
                          p_searcher,
                          M_EXISTING_MOBILE_SEARCHER);
```


2.1.3 User Data Areas

Each VMI object has a user data block that is solely for application use. The size of the user data block is configurable via the defines in **cbme.h**. The default size is one byte. User data is stored and retrieved via the *object*_Set_User_Data and *object*_Get_User_Data functions. See Section 14.3 (page 172) for details on these two functions.

2.2 CBME Interrupts

The interrupt service (code provided by Morphics Technology) is a 'C' function that will be called from the CBME hardware interrupt routine.

The CBME has two external interrupt pins called the High and Low Priority Interrupt Pins. These interrupt pins are used as inputs to a microprocessor to notify it of CBME events. The events associated with each interrupt are shown in Table 2-3:

Table 2-3: CBME Interrupts

Interrupt	Interrupt Source(s)
High Priority Interrupt	Searcher and preamble detection engine results
Low Priority Interrupt	Finger results and error conditions

There is an option to merge the two interrupts; when this option is selected, all interrupt sources will be directed to the High Priority Interrupt pin.

	Merged Interrupts	Separate Interrupts
Advantages	• The CBME only takes up one interrupt pin on the microprocessor • Only one interrupt routine required	• Improved processing performance because each of the two interrupt routines checks for a subset of total interrupt events.
Disadvantages	• Degraded processing performance in the single interrupt routine in that it has to check for all interrupt events (including errors)	• CBME takes up two interrupt pins on the microprocessor • Two interrupt routines required

The VMI library call, *CBME_New*, is used to notify the VMI whether or not interrupts have been merged.

2.3 *Real Time Operating System (RTOS) Interface*

Any preemptive, multitasking operating system that supports task message queues can be used with the Morphics VMI library.

2.3.1 RTOS Restrictions

The VMI is designed to work within an RTOS environment presuming the following restrictions:

- The RTOS must support task message queues with the ability to create, write to, and read from a message queue.
- The message queue send service must be callable from an interrupt routine
- All CBME object functions are called from a single task
- All Uplink, Combiner, Finger and Searcher, and Searcher DSM object functions must be called from a single task
- All Preamble Detection Engine object functions are called from a single task
- All Downlink object functions must be called from a single task

Figure 2-2 : Legal Task Access (4 tasks)

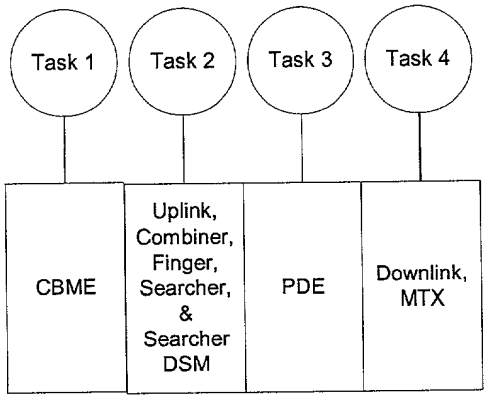


Figure 2-3 : Legal Task Access (1 task)

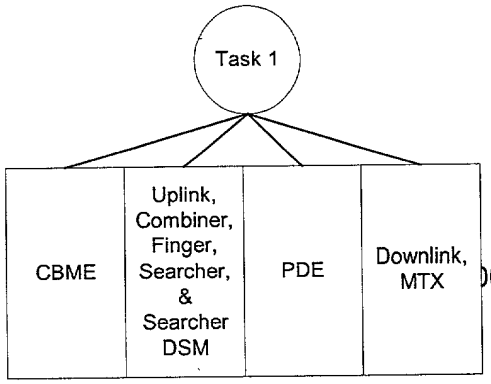
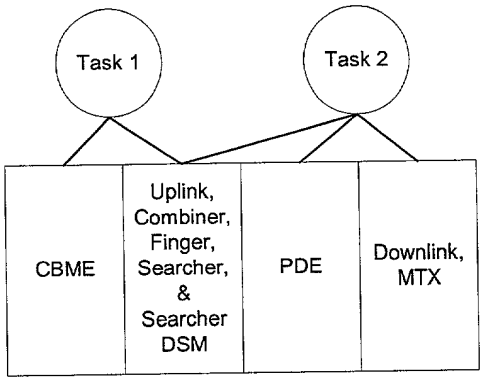


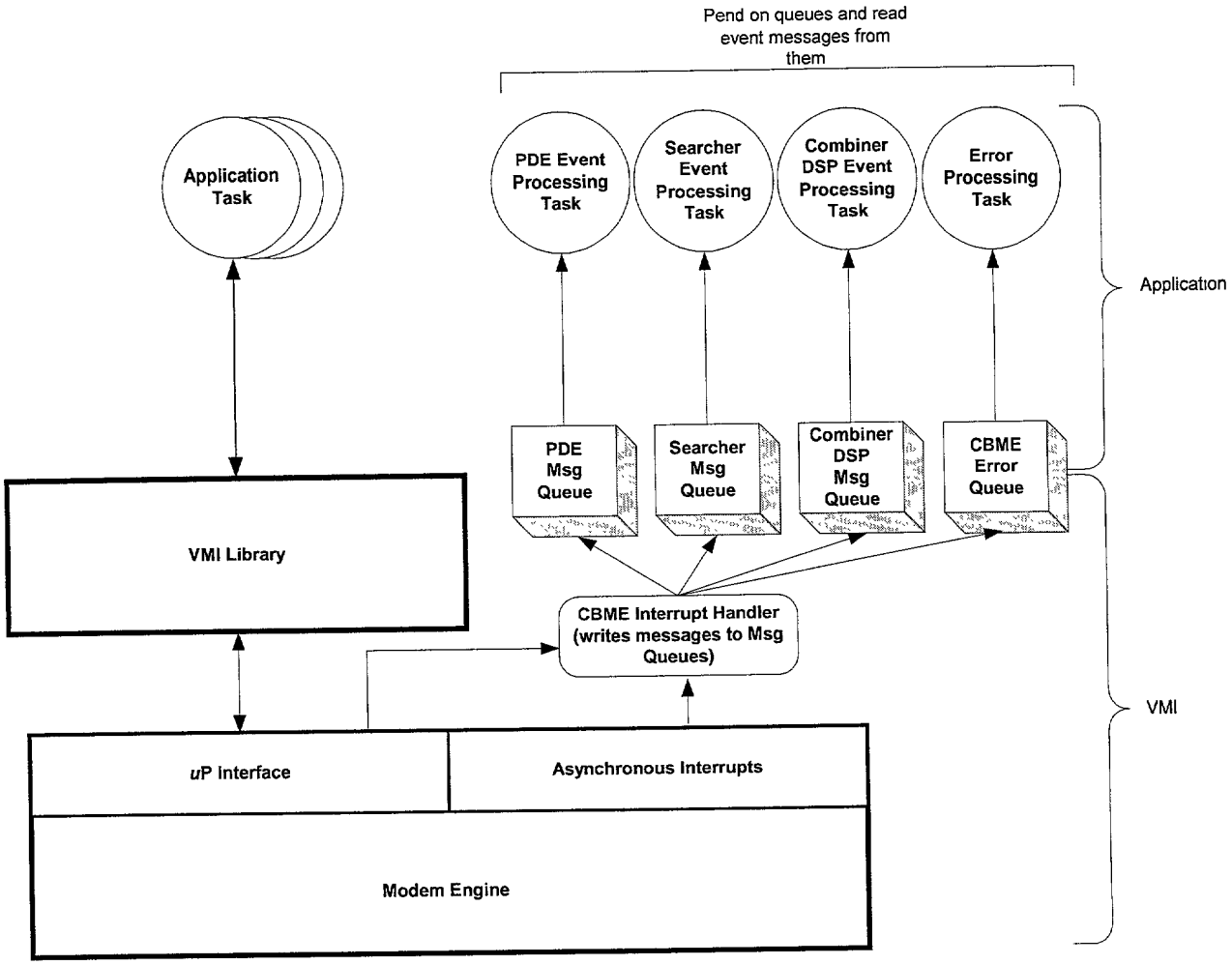
Figure 2-4 : Illegal Task Access (Cannot have two tasks accessing same object)



2.3.2 RTOS Interface Model

The CBME RTOS interface model is shown in Figure 2-5:

Figure 2-5 : CBME RTOS Interface Model



The VMI, at initialization, will create the message queues. The application will create the event processing tasks that pend on the message queues. The CBME interrupt routine will write messages to the message queues, which will wake up the pending task to process the event.

As can be seen in Figure 2-5, the VMI generates four message queues that are used to signify CBME events. These message queues are:

Table 2-4: Event Message Queues

Message Queue	Events Signified by Messages
PDE Message Queue	• PDE energies.
Searcher Message Queue	• Searcher energies.
Combiner DSP Message Queue	• Combiner DSP messages (e.g. finger energies) • Response to Finger Offset requests
Error Message Queue	• Internal CBME Errors

2.3.3 User-Supplied Message Queue Functions

When the CBME is initialized and running, events (e.g. Searcher energy results) will occur, and the application needs to be made aware of these events. The RTOS message queues shown in Figure 2-5 are the mechanism for the VMI library to notify the application of events.

The VMI library initialization code will create the message queues shown in Figure 2-5. The CBME interrupt routine will write messages to these queues. The application will create tasks that pend on these queues, and these task will 'wake up' when a message needs to be processed.

In order for the VMI library to create and write to message queues, it requires two functions:

VMI_Msg_Queue_Create
VMI_Msg_Queue_Send

The function prototypes are shown in Table 2-5. The above functions are higher-level wrappers that will apply to any RTOS that supports message queues. The user will need to fill in the body of these functions to support the specific RTOS they are using. An example of a VxWorks implementation is shown in Section 14.5.

The function prototypes for the VMI message queue functions are located in **m_rtos.h**. The file, **m_rtos.c**, contains the function definitions. The default supplied with the VMI library is code that supports VxWorks. If using a different RTOS, the code in m_rtos.c will need to be modified as described in this section.

Remember, these are functions that only internal VMI functions call; the application software never calls these functions directly. However, the application must create tasks that pend on the message queues and will read event messages from them.

Table 2-5: Required RTOS Message Queue Functions

Function	Description
`UINT16 VMI_Msg_Queue_Create( VMI_MSG_Q_ENUM q_type, UINT16 max_msg_length, UINT16 max_msgs);`	This function is used by the VMI to create an event message queue. *CBME_New* (see Section 3.1.1) will call *VMI_Msg_Queue_Create* once per message queue that is required. *q_type* is an enumerated type that identifies the queue. *max_msg_length* is the maximum length, in bytes, of a message that will be written to this queue. *max_msgs* is the maximum number of messages that can be stored in the queue. There are four defines in m_rtos.h: PDE_QUEUE_MAX_MSG_COUNT SEARCHER_QUEUE_MAX_MSG_COUNT COMBINER_DSP_QUEUE_MAX_MSG_COUNT ERROR_QUEUE_MAX_MSG_COUNT These are the values the VMI will use for *max_msgs*. If you experience queue overflow errors, then make these defines larger and rebuild the VMI library. The function returns an error code of either M_SUCCESS or M_RTOS_MSG_QUEUE_CREATE_ERROR.
`UINT16 VMI_Msg_Queue_Send( VMI_MSG_Q_ENUM q_type, UINT32 *p_msg, UINT16 msg_length);`	This function is used by the VMI to send messages to the event message queues. *q_type* is the enumerated type that identifies the queue. *p_msg* is the pointer to the message being written. *msg_length* is the length, in bytes, of the message. The function returns an error code of either M_SUCCESS or M_RTOS_MSG_QUEUE_SEND_ERROR.

2.3.4 Queue Message Formats

This section describes the format of messages written into the event message queues by the CBME interrupt. For all queues, the messages are sent as an array of 32-bit words.

The first word of any message is always a header that is formatted as follows:

Header Word (Word 1)

31 – 16	15 - 0
Msg Type	Length

Field	Description
Msg Type	Identifier for the message
Length	Length of this message, in 32-bit words, including the header word.

The description of the messages that will be sent into the event queues by the VMI are described in the following sections:

Searcher Events	-	Section 6.2
Preamble Detection Engine Events	-	Section 8.2
Combiner DSP Events	-	Section 11.2
CBME Error Events	-	Section 2.5

2.4 *VMI Error Checking*

VMI error checking can be enabled or disabled via a #define in **cbme.h**. If the following line in this file:

```
#define VMI_ENABLE_ERROR_CHECKING
```

is commented out, then error checking is disabled. If this line is not commented out, then error checking is enabled.

When error checking is enabled, all VMI functions return a valid error code, either by the return value or by a pointer to an input parameter. A summary of the error codes is in Section 14.1.

Additionally, to avoid having to check an error code after each VMI function call, a VMI function, *VMI_Process_Error*, is called whenever a VMI function returns an error. The user may customize this function to suit their application (e.g. serial output, file logging, etc.).

VMI_Process_Error is found in **m_error.c**. The protoype is:

```
void VMI_Process_Error(UINT16    error_code,
                       char      *p_buf,
                       int       line_num);
```

error_code	– 16-bit VMI error code
p_buf	– string containing the name of the VMI module where the error occurred
line_num	– line number in the module where the error function was called from

2.5 *CBME Error Events*

This section describes the error events generated by the CBME. These events are reported via the Error Message Queue. Refer to Section 2.3.2 to for how this queue is created and accessed.

2.5.1 Error Queue Messages

This section describes the format of the Error messages that will be sent by the VMI to the Error Message Queue. The error messages are system-level messages; they do not apply to a specific object. The error messages consist of only the header word; there is no other data associated with the message. Thus, the format for all Error Queue messages is:

Word 1 (Header Word)

31 - 16	15 - 0
Msg Type (error code)	Length (always 1)

Error Messages (Msg Type)	Description
SEARCHER_QUEUE_OVERFLOW_MSG	Searcher Message Queue has overflowed; data has been lost.
PDE_QUEUE_OVERFLOW_MSG	PDE Message Queue has overflowed; data has been lost.
COMBINER_DSP_QUEUE_OVERFLOW_MSG	Combiner DSP Message Queue has overflowed; data has been lost.

3 CBME

A single microprocessor can control one or more CBME ASICs by creating a CBME object for each ASIC. All other objects (Uplinks, Downlinks, Combiners, Fingers, etc.) are associated with a specific CBME object.
The normal calling sequence to initialize the CBME is:

CBME_New()	Allocate a CBME object.
CBME_Scanchain_Write()	Write out RAM scanchain
CBME_Scanchain_Write()	Write out Register scanchains
CBME_Get_Resource_Attributes()	Retrieve the CBME resources available
CBME_Set_Mobile_Resources()	Configure the number of mobiles supported
CBME_Set_Search_Periodicity()	Set the searcher periodicity
CBME_Set_Searcher_Energy_Scaling()	Set the searcher energy report scaling
CBME_Set_DSM_Subchip_Phase()	Set the DSM subchip phase
CBME_Set_PDE_Num_Slots()	Set the number of access slots for the Preamble Detection Engine
CBME_Get_CGU_List()	Retrieve the CGUs that are available
CBME_Get_Downlink_Slot_Format_List()	Retrieve list of downlink slot formats available
CBME_Get_Downlink_Field_List()	Retrieve list of multiplexed transmission fields
CBME_Get_Uplink_Slot_Format_List()	Retrieve list of uplink slot formats available

3.1 CBME Methods

3.1.1 CBME_New

Prototype

UINT16 **CBME_New**(CBME *p_cbme,
UINT32 *p_base_address,
UINT8 merge_interrupt_action);

Description

Allocates a new CBME.

Input Parameters

p_cbme	pointer to CBME that is being allocated
p_base_address	pointer to the base address of the CBME *u*P interface.
merge_interrupt_action	See *CBME Interrupts* (Section 2.2).

Define Value	Description
M_SEPARATE_INT	Do not merge interrupts
M_MERGE_INT	Merge interrupts

Restrictions

This function must be called before any other VMI functions.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.2 CBME_Scanchain_Write

Prototype

```
UINT16 CBME_Scanchain_Write(CBME    *p_cbme,
                            UINT16  count,
                            UINT32  *p_scanchain_data,
                            UINT16  scanchain_type);
```

Description

This function must be called twice, once for the RAM Scanchain, and once for the Register Scanchains.

The CBME has 32 scanchains. One scanchain is used to initialize the CBME on-board RAMs and is downloaded separately. The other 31 scanchains are Register scanchains and configure various internal registers on the CBME. The CBME cannot operate correctly until its scanchains are downloaded.

Morphics provides utility programs that generate the binary images for the scanchain downloads. This function is passed a pointer to the scanchain image generated by the utility and then performs the download. The *scanchain_type* field is used to distinguish whether the RAM scanchain is being downloaded or the Register scanchains are being downloaded.

Scanchains can be read back via the *CBME_Scanchain_Read* function.

Input Parameters

p_cbme	pointer to CBME
count	number of 32-bit words in the scanchain data
p_scanchain_data	pointer to array containing scanchain data
scanchain_type	M_RAM_SCANCHAIN, M_REG_SCANCHAINS

Restrictions

CBME_New must be called first.

The scanchain download sequence must be:

1) download RAM scanchain
2) download Register scanchains

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.3 CBME_Scanchain_Read

Prototype

UINT16 **CBME_Scanchain_Read**(CBME *p_cbme,
UINT16 count,
UINT32 *p_scanchain_data,
UINT16 scanchain_type);

Description

See description for *CBME_Scanchain_Write*. This function reads either the RAM scanchain or the Register scanchains

Input Parameters

p_cbme	pointer to CBME
count	number of 32-bit words to read from scanchain
p_scanchain_data	pointer to array where scanchain data will be written.
scanchain_type	M_RAM_SCANCHAIN, M_REG_SCANCHAINS

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.4 CBME_Free

Prototype

UINT16 **CBME_Free**(CBME *p_cbme);

Description

Deallocates a CBME.

Input Parameters

p_cbme pointer to CBME object being deallocated.

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.5 CBME_Set_Mobile_Resources

Prototype

```
UINT16 CBME_Set_Mobile_Resources(CBME   *p_cbme,
                                 UINT16 finger_block_size,
                                 UINT16 num_mobiles);
```

Description

Configures the CBME for:

(a) the number of mobiles that can be supported
(b) the tracking finger block size for each mobile.

This function should be called after calling CBME_Get_Resource_Attributes() which will return *max_fingers*, the maximum number of tracking fingers supported by the CBME at its input clock rate. Using *max_fingers*, a determination can be made (based on system requirements) on how many mobiles to support, and for each mobile, what will be the initial number of tracking fingers available to it.

See Section 14.2 (Page 170) for a detailed description of how this function affects performance and resources.

Input Parameters

p_cbme	pointer to CBME
finger_block_size	minimum number of tracking fingers allocated per mobile Valid Range: 4, 6, or 8
num_mobiles	number of mobiles to support. Num_mobiles ≤ (max_fingers / finger_block_size) *max_fingers is a resource attribute (see Section 3.1.12.1)

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.6 CBME_Set_Search_Periodicity

Prototype

UINT16 **CBME_Set_Search_Periodicity**(CBME *p_cbme,
UINT16 timer_constant);

Description

This function sets the search period for all searchers under the CBME. The search period is nominally 50ms, and is defined by the following formula:

timer_constant = (input_chipping_rate * search_period) / 256

For example, presume:
desired search period = 50ms
input chipping rate = 3.84 Mcps:

timer_constant = (3.84E6 * 50E-3) / 256 = 750

The minimum duration for the timer setting should be such that it does not restart a new search before the completion of the previous search.

The maximum possible time for completion of a search is the maximum of the following expression using parameters set in the Searchers DSM (in number of chips):

(integration_length * pdi_length) * (search_resolution / 2) * (search_window)

See Section 6.2 for a description of the DSM.

Input Parameters

p_cbme	pointer to CBME
timer_constant	0x0001 to 0xffff

Restrictions

CBME_New must be called first. This function must be called before any searchers are allocated (see Searcher_New).

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.7 CBME_Set_Searcher_Energy_Scaling

Prototype

UINT16 **CBME_Set_Searcher_Energy_Scaling**(CBME *p_cbme,
UINT16 scale_value);

Description

Internally, the CBME generates a 32-bit search result value. However, only 12 bits are reported to the microprocessor. This function sets the range of energy bits to report. This setting affects all searchers under the CBME.

Input Parameters

p_cbme pointer to CBME

scale_value Effectively, this field indicates how many bits to left shift the searcher energy before reporting to the microprocessor.

Search Energy Reported	scale_value (define)
$Energy_{31-20}$	M_SEARCHER_SCALE_31_20
$Energy_{30-19}$	M_SEARCHER_SCALE_30_19
$Energy_{29-18}$	M_SEARCHER_SCALE_29_18
$Energy_{28-17}$	M_SEARCHER_SCALE_28_17
↓	↓
$Energy_{12-1}$	M_SEARCHER_SCALE_12_1
$Energy_{11-0}$	M_SEARCHER_SCALE_11_0

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.8 CBME_Set_DSM_Subchip_Phase

Prototype

```
UINT16 CBME_Set_Subchip_Phase(
            CBME                        *p_cbme,
            M_SUBCHIP_PHASE_TYPE        subchip_phase);
```

Description

This function pertains to the Dwell State Machine (DSM) used by Searchers; it configures the ½ chip searches. These settings apply to all Searcher DSM's.

Input Parameters

p_cbme	pointer to CBME
subchip_phase	see Section 3.1.8.1

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.8.1 M_SUBCHIP_PHASE_TYPE

```
typedef struct subchip_phase_struct
{
        UINT16      half_phase_low;
        UINT16      half_phase_high;

} M_SUBCHIP_PHASE_TYPE ;
```

Field	Description
half_phase_low	Valid Range: 0 to 3 (eighth chips)
half_phase_high	Valid Range: 4 to 7 (eighth chips)

3.1.9 CBME_Set_PDE_Num_Slots

Prototype

```
UINT16 CBME_Set_PDE_Num_Slots(      CBME        *p_cbme,
                                    UINT16      num_access_slots,
                                    UINT16      num_access_slot_sets);
```

Description

Configures the number of Preamble Detection Engine access slots and the number of access slot sets. See *Preamble Detection Engine* (Section 8) for a detailed explanation of this function. This function must be called before any Preamble Detection Engines are allocated.

Input Parameters

p_cbme	pointer to CBME
num_access_slots	number of desired access slots. Valid Range: 1 to *max_pdes**
	*total number of preamble detection engines supported derived from *CBME_Get_Resource_Attributes()*.
num_access_slot_sets	*max_pdes* ≥ (num_access_slots * num_access_slot_sets)

Restrictions

CBME_New must be called first.

CBME_Get_Resource_Attributes should be called to determine the maximum number of Preamble Detection Engines that can be allocated.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.10 CBME_Perform_Self_Tests

Prototype

UINT16 **CBME_Perform_Self_Tests**(CBME *p_cbme, UINT16 selfTest);

Description

Perform self-tests and returns result via *VMI_Event_Notify* function.

Input Parameters

p_cbme pointer to CBME
selfTest self test to perform (TBD)

Restrictions

CBME_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.11 CBME_Get_Mobile_Resources

Prototype

UINT16 **CBME_Get_Mobile_Resources**(CBME *p_cbme,
UINT16 *p_finger_block_size,
UINT16 *p_num_mobiles);

Description

Retrieves the values set in *CBME_Set_Mobile_Resources*

Input Parameters

p_cbme pointer to CBME
p_finger_block_size pointer to where the finger block size is written
num_mobiles pointer to where the number of mobiles is written

Restrictions

CBME_Set_Mobile_Resources must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.12 CBME_Get_Resource_Attributes

Prototype

UINT16 **CBME_Get_Resource_Attributes(**
CBME *p_cbme,
CBME_RESOURCE_ATTRIB *p_resource_attrib);

Description

Determines the resources available within the CBME ASIC.

Input Parameters

p_cbme	pointer to CBME
p_resource_attrib	pointer to where resource attributes will be written. See Section 3.1.12.1 for a description of the resource attributes.

Restrictions

CBME_New must be called first.

CBME_Scanchain_Write must be called twice – once for the RAM scanchain, and once for the Register scanchains.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.12.1 CBME_RESOURCE_ATTRIB

The 'C' structure is:

```
typedef struct  cbme_resource_attrib_struct
{
  /* SEARCHERS */
  UINT16  max_searchers;
  UINT16  num_new_mobile_searchers_allocated;
  UINT16  num_existing_mobile_searchers_allocated;
  UINT16  num_new_mobile_searchers_running;
  UINT16  num_existing_mobile_searchers_running;

  /* DSMs */
  UINT16  max_dsm;
  UINT16  num_dsm_allocated;

  /* PREAMBLE DETECTION ENGINES */
  UINT16  max_pdes;
  UINT16  num_pdes_allocated;

  /* PREAMBLE DETECTION ENGINE ANTENNAS */
  UINT16  max_pde_antennas;
  UINT16  num_pde_antennas_allocated;

  /* TRACKING FINGERS */
  UINT16  max_fingers;
  UINT16  num_fingers_allocated;
  UINT16  num_fingers_running;
  UINT16  num_channels_per_finger;
  UINT16  num_reserve_pairs_available;

  /* COMBINERS */
  UINT16  max_combiners;
  UINT16  num_combiners_allocated;
  UINT16  num_combiners_running;

  /* UPLINKS */
  UINT16  num_uplinks_allocated;
  UINT16  max_uplink_slot_formats;
  UINT16  max_uplink_antenna_port;
  UINT16  uplink_antenna_buffer_size;

  /* DOWNLINKS */
  UINT16  max_downlinks;
  UINT16  max_downlink_slot_formats;
  UINT16  num_downlinks_allocated;
  UINT16  num_downlinks_running;
```



```
/* MTX */
UINT16  num_mtx_allocated;

/* Diversity */
UINT16  num_diversity_allocated;

/* CGUs */
UINT16  num_cgus_allocated;

} CBME_RESOURCE_ATTRIB;
```

Table 3-1: CBME Resource Attributes

CBME Resource Attributes	Description
Searcher Resources	
max_searchers	Maximum number of Searchers that can be allocated.
num_new_mobile_searchers_allocated	Number of 'new mobile' Searchers that have been allocated.
num_existing_mobile_searchers_allocated	Number of 'existing mobile' Searchers that have been allocated.
num_new_mobile_searchers_running	Number of 'new mobile' Searchers that are currently running.
num_existing_mobile_searchers_running	Number of 'existing mobile' Searchers that are currently running.
DSM Resources	
max_dsm	Maximum number of DSMs that can be allocated.
num_dsm_allocated	Number of DSMs that have been allocated.
Preamble Detection Engine Resources	
max_pdes	Maximum number of Preamble Detection Engines that can be allocated.
num_pdes_allocated	Number of Preamble Detection Engines that have been allocated.
Preamble Detection Engine Antenna Resources	
max_pde_antennas	Maximum number of Preamble Detection Engine Antennas that can be allocated.
num_pde_antennas_allocated	Number of Preamble Detection Engine Antennas that have been allocated.
Finger Resources	
max_fingers	Maximum number of Fingers that can be allocated.
num_fingers_allocated	Number of Fingers that have been allocated.
num_fingers_running	Number of Fingers that are currently running.
num_channels_per_finger	Number of logical channels supported by a Finger object.

CBME Resource Attributes	Description
num_reserve_pairs_available	Number of reserve finger pairs are available (see Section 14.2)
Combiner Resources	
max_combiners	Maximum number of Combiners that can be allocated.
num_combiners_allocated	Number of Combiners that can be allocated.
num_combiners_running	Number of Combiners that are currently running.
max_fingers_per_combiner	Maximum number of Fingers that each Combiner can have added to it.
Uplink Resources	
num_uplinks_allocated	Number of Uplinks that have been allocated.
max_uplink_slot_formats	Number of uplink slot formats in the Uplink Slot Format List.
max_uplink_antenna_port	Maximum antenna port number that can be used by searchers and fingers. Antenna ports are numbers range from 0 to *max_uplink_antenna_port*.
uplink_antenna_buffer_size	The size of an individual uplink antenna port buffer in chips.
Downlink Resources	
max_downlinks	Maximum number of Downlink objects that can be allocated.
max_downlink_slot_formats	Number of downlink slot formats in the Downlink Slot Format List.
num_downlinks_allocated	Number of Downlink that have been allocated.
num_downlinks_running	Number of Downlink that are running.
MTX Resources	
num_mtx_allocated	Number of MTXs that have been allocated.
Diversity Resources	
num_diversity_allocated	Number of Downlink diversity channels that have been allocated.
CGU Resources	
num_cgus_allocated	Number of CGUs that have been allocated.

3.1.13 CBME_Get_CGU_List

Prototype

UINT16 **CBME_Get_CGU_List**(CBME *p_cbme,
M_CGU_LIST_TYPE *p_cgu_list);

Description

The on-chip CGUs are configured during scanchain download. This function is called after the scanchain has been downloaded to get a list of the on-chip CGUs that are supported.

Note that the CGU object (described in Section 4) is based upon one of the CGUs in the CGU List. There can be many variations of CGU objects based upon a single CGU from the CGU List.

Input Parameters

p_cbme	pointer to CBME
p_cgu_list	pointer to list of on-chip CGUs. See Section 3.1.13.1.

Restrictions

CBME_New must be called first.

Scanchains must be downloaded before calling this function (*CBME_Scanchain_Write).*

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.13.1 M_CGU_LIST_TYPE

This data-type is used by *CBME_Get_CGU_List*; after calling the function, the structure below will be filled in and can be utilized. Note that the CGU list is an array of the same structure used in *CGU_Set_Static_Attributes* (see Section 4.1.3.1). All the fields are valid after calling *CBME_Get_CGU_List* except the *code_number* field, which will always be 0 (this field is set for each CGU object when calling *CGU_New*).

```
typedef struct cgu_list_struct
{
  UINT16                          num_on_chip_cgus;

  M_CGU_STATIC_ATTRIB_TYPE   cgu[M_MAX_CGU_PER_CBME];

} M_CGU_LIST_TYPE;
```


3.1.14 CBME_Get_Downlink_Field_List

Prototype

UINT16 **CBME_Get_Downlink_Field_List** (
CBME *p_cbme,
M_DOWNLINK_FIELD_LIST_TYPE *p_field_list);

Description

The CBME supports one or more multiplexed transmission fields, and these are configured during scanchain download. This function retrieves a list of strings that correspond to name of each multiplexed field supported. The indexes of the returned list correspond to the indexes used in the *field_power_levels* field in the *Downlink Static Attributes Structure* (Section 12.1.3.1).

Input Parameters

p_cbme	pointer to CBME
p_field_list	pointer to list of supported downlink multiplexed field types. See Section 3.1.14.1.

Restrictions

CBME_New must be called first.

Scanchains must be downloaded before calling this function (*CBME_Scanchain_Write).*

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.14.1 M_DOWNLINK_FIELD_LIST_TYPE

```
typedef struct downlink_field_list_struct
{
  UINT16  num_defined_fields;

  char    field_name[M_NUM_DOWNLINK_POWER_TYPES][M_MAX_FIELD_NAME_LENGTH];

  UINT16  min_power;

  UINT16  max_power;

  UINT16  fractional_range;

} M_DOWNLINK_FIELD_LIST_TYPE
```

Field	Description
num_defined_fields	The number of multiplexed fields in the list. For example, if this value is 9, then indexes 0 – 8 are valid in the *field_name* array.
field_name	Array of strings describing the supported field types. The first index is the string within the array, the second index is the max length of each string within the array.
min_power	Minimum power level in whole dBs (e.g. 6) for all powers.
max_power	Maximum power level in whole dBs (e.g. 45) for all powers.
fractional_range	The fractional resolution value of a field power level. For example, if *fractional_range* = 8, then field power levels can be set at a 1/8 dB resolution: 0 : 0/8 dB 1 : 1/8 dB 2 : 2/8 dB . . 7 : 7/8 dB [max allowed value] Range applies to all field powers.

3.1.15 CBME_Get_Downlink_Slot_Format_List

Prototype

```
UINT16 CBME_Get_Downlink_Slot_Format_List(
            CBME                                    *p_cbme,
            M_DOWNLINK_SLOT_FORMAT_LIST_TYPE        *p_slot_format_list);
```

Description

The supported CBME transmitter channels are configured via the scanchain download. This function retrieves a list of strings that correspond to the name of each channel supported.

The index of the string for the desired channel is used as the value for *slot_format_index* in the Downlink Static Attributes structure (see Section 12.1.3.1) to select the channel type for the Downlink.

Input Parameters

p_cbme	pointer to CBME
p_slot_format_list	pointer to list of supported transmitter channels. See Section 3.1.15.1.

Restrictions

CBME_New must be called first.

Scanchains must be downloaded before calling this function (*CBME_Scanchain_Write).*

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.15.1 M_DOWNLINK_SLOT_FORMAT_LIST_TYPE

There are two structures used to describe the downlink slot formats. The M_DOWNLINK_SLOT_FORMAT is the descriptor for a specific slot. The M_DOWNLINK_SLOT_FORMAT_LIST_TYPE contains the number of slot formats in the list and an array of M_DOWNLINK_SLOT_FORMAT.

```
typedef struct downlink_slot_struct
{
   char     name[M_MAX_SLOT_NAME_LENGTH];

   UINT8    channel_type;

   UINT16   spreading_factor;

} M_DOWNLINK_SLOT_FORMAT;
```

Field	Description
name	String name of slot format. "NULL" if not used.
channel_type	M_NOT_SYNC_CHANNEL or M_PRIMARY_SYNC_CHANNEL or M_SECONDARY_SYNC_CHANNEL
spreading_factor	M_SPREADING_FACTOR_4 or M_SPREADING_FACTOR_8 or M_SPREADING_FACTOR_16 or M_SPREADING_FACTOR_32 or M_SPREADING_FACTOR_64 or M_SPREADING_FACTOR_128 or M_SPREADING_FACTOR_256

```
typedef struct m_downlink_slot_format_list_struct
{
   UINT16   num_defined_slot_formats;

   M_DOWNLINK_SLOT_FORMAT   slot[M_NUM_DOWNLINK_SLOT_FORMATS];

} M_DOWNLINK_SLOT_FORMAT_LIST_TYPE;
```

Field	Description
num_defined_slot_formats	Number of slot formats in the list
slot	Array of slot format descriptions

The following function demonstrates how to search the list looking for a particular slot format The input is a string name of the desired channel. The function returns the slot format index if the channel is found, else – 1:

CBME cbme;

```
int  Find_Tx_Channel(char *p_desired_channel)
{

   /* for each slot format in the list */
   for(i = 0;
       i < cbme.downlink_slot_format_list.num_defined_slot_formats;
       ++i)
   {
      /* if a match */
      if ((strcmp(cbme.downlink_slot_format_list.slot[i].name,
           p_desired_channel) == 0))
      {
         return (i);  /* return slot format index of channel */
      }
   }

   return (-1); /* -1 indicates channel not found */
}
```


3.1.16 CBME_Get_Uplink_DPCCH_Slot_Format_List

Prototype

UINT16 **CBME_Get_Uplink_DPCCH_Slot_Format_List(**
CBME *p_cbme,
M_UPLINK_DPCCH_SLOT_FORMAT_LIST_TYPE *p_dpcch_slot_format_list);

Description

The supported CBME receiver DPCCH slot formats are configured via the scanchain download. This function retrieves a list of strings that correspond to the name of each Uplink DPCCH slot format supported.

The index of the string for the desired slot format is used as the value for *slot_format* field in the *Uplink_Set_DPCCH_Slot_Format* function (see Section 5.1.3).

Input Parameters

p_cbme	pointer to CBME
p_slot_format_list	pointer to list of supported transmitter channels. See Section 3.1.16.1

Restrictions

CBME_New must be called first.

Scanchains must be downloaded before calling this function (*CBME_Scanchain_Write).*

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.16.1 M_UPLINK_DPCCH_SLOT_FORMAT_LIST_TYPE

```
typedef struct uplink_slot_struct
{
   char     name[M_MAX_SLOT_NAME_LENGTH];

} M_UPLINK_DPCCH_SLOT_FORMAT;
```

Field	Description
name	String name of uplink DPCCH slot format. "NULL" if not used.

```
typedef struct m_uplink_slot_format_list_struct
{
   UINT16  num_defined_slot_formats;

   M_UPLINK_DPCCH_SLOT_FORMAT  slot[M_NUM_UPLINK_SLOT_FORMATS];

} M_UPLINK_DPCCH_SLOT_FORMAT_LIST_TYPE;
```

Field	Description
num_defined_slot_formats	Number of slot formats in the list
slot	Array of slot formats in the list

3.1.17 CBME_Get_Searcher_Energy_Scaling

Prototype

UINT16 **CBME_Get_Searcher_Energy_Scaling**(CBME *p_cbme,
UINT16 *p_scale_value);

Description

Retrieves the searcher energy-scaling factor. See CBME_Set_Searcher_Energy_Scaling for description of the scaling factors.

Input Parameters

p_cbme	pointer to CBME
p_scale_value	pointer where scale factor is written.

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.18 CBME_Get_DSM_Subchip_Phase

Prototype

```
UINT16 CBME_Get_Subchip_Phase(
        CBME                        *p_cbme,
        M_SUBCHIP_PHASE_TYPE        *p_subchip_phase);
```

Description

This function retrieves the value of the subchip phases set in *CBME_Set_DSM_Subchip_Phase* (or the default values if no values have been set).

Input Parameters

p_cbme	pointer to CBME
p_subchip_phase	pointer to where values will be written. See Section 3.1.8.1 for definition of data type.

Restrictions

CBME_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.19 CBME_Get_Search_ Periodicity

Prototype

UINT16 **CBME_Get_Search_Periodicity**(CBME *p_cbme,
UINT16 *p_timer_constant)

Description

This function retrieves the values set in *CBME_Set_Search_Periodicity.*

Input Parameters

p_cbme	pointer to CBME
p_timer_constant	pointer to where timer constant will be written

Restrictions

CBME_Set_Search_Periodicity must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.20 CBME_Get_PDE_Num_Slots

Prototype

UINT16 **CBME_Get_PDE_Num_Slots**(CBME *p_cbme,
UINT16 *p_num_access_slots,
UINT16 *p_num_access_slot_sets)

Description

This function retrieves the values set in *CBME_Set_PDE_Num_Slots*

Input Parameters

p_cbme	pointer to CBME
p_num_access_slots	pointer to where value is written
p_num_access_slot_sets	pointer to where value is written

Restrictions

CBME_Set_PDE_Num_Slots must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.21 CBME_Get_Software_Version

Prototype

UINT16 **CBME_Get_Software_Version**(UINT16 *p_versionNumber);

Description

Determines the software version number of the CBME VMI library.

Input Parameters

p_versionNumber	pointer to where software version number will be written The most significant 8 bits are the major version number, the least significant 8 bits are the minor version number. For example, 0x0106 corresponds to Version 1.6.

Restrictions

None

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.22 CBME_Get_Hardware_Version

Prototype

UINT16 **CBME_Get_Hardware_Version**(CBME *p_cbme,
UINT16 *p_versionNumber);

Description

Determines the hardware version number of the CBME device.

Input Parameters

p_cbme	pointer to CBME
p_versionNumber	pointer to where hardware version number will be written The most significant 8 bits are the major version number, the least significant 8 bits are the minor version number. For example, 0x0106 corresponds to Version 1.6.

Restrictions

CBME_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

3.1.23 CBME_Set_User_Data

Prototype

```
UINT16 CBME_Set_User_Data(CBME    *p_cbme,
                          UINT16  index,
                          UINT16  length,
                          UINT8   *p_data);
```

Description

See Section 14.3.1 for a description of this function.

3.1.24 CBME_Get_User_Data

Prototype

```
UINT16 CBME_Get_User_Data(CBME  *p_cbme,
                          UINT16  index,
                          UINT16  length,
                          UINT8   *p_data);
```

Description

See Section 14.3.2 for a description of this function.

4 CGU

The VMI CGU object is based on one of the on-chip CGUs configured via the scanchain and downloaded with *CBME_Scanchain_Write*. There can be many CGU objects based upon a single CGU from the CGU List.

The on-chip CGUs are object-specific in that each on-chip CGU only works with one type of object. Specifically, each of the following VMI objects must have an associated CGU:

Uplink
Searcher ('new mobile')
Preamble Detection Engine Antennas
Downlink

The *CBME_Get_CGU_List* function is used to determine the number of on-chip CGU's, their configurations, and the object-type they support. This function is only valid after the scanchains have been downloaded.

An example code fragment to use the CGU object is:

```
     CBME_New(p_cbme, p_base_address, M_MERGE_INT);
     CBME_Scanchain_Write(p_cbme,                    /* RAM scanchain
*/
                          BUFFER_SIZE,
                          p_ram_scanchain_data,
                          M_RAM_SCANCHAIN);

     CBME_Scanchain_Write(p_cbme,                    /* Register scanchains
*/
                          BUFFER_SIZE,
                          p_all_scanchain_data,
                          M_ALL_SCANCHAINS);

     /* get the list of on-chip CGUs */
     CBME_Get_CGU_List(p_cbme, p_cgu_list);

     /* parse list, determine CGU parameters and object associations */
     /* presume that an Uplink CGU is at index 4 in the list, and is */
     /* configured for the desired standard                          */

     /* allocate a new CGU, associate with index 4 in the CGU list,  */
     /* assign a code number                                         */
     CGU_New(p_cbme, p_cgu, 4, 0x12345678);

     /* create an Uplink and associate with the CGU */
     Uplink_New(p_cbme, p_uplink, p_cgu);

     /* if desired, get the attributes of this CGU */
     CGU_Get_Static_Attributes(p_cbme,  p_cgu_static_attrib);
```


4.1 CGU Methods

CBME_New must be called prior to any CGU Methods. This restriction is not repeated for each function description.

4.1.1 CGU_New

Prototype

```
UINT16 CGU_New(CBME   *p_cbme,
               CGU    *p_cgu,
               UINT16  cgu_index,
               UINT32  code_number);
```

Description

Allocates a new CGU.

Input Parameters

p_cbme	pointer to CBME
p_cgu	pointer to CGU being allocated
cgu_index	index into on-chip CGU list that was generated by calling *CBME_Get_CGU_List*.
code_number	code number for the standard associated with this CGU

Restrictions

None

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

4.1.2 CGU_Free

Prototype

UINT16 **CGU_Free**(CGU *p_cgu);

Description

Deallocates a CGU.

Input Parameters

p_cgu	pointer to CGU being deallocated

Restrictions

Cannot free a CGU that is in use by any other objects (attach count > 0)

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

4.1.3 CGU_Get_Static_Attributes

Prototype

UINT16 **CGU_Get_Static_Attributes(**
CGU *p_cgu,
CGU_STATIC_ATTRIB_TYPE *p_cgu_static_attrib);

Description

Gets the CGU static attributes.

Input Parameters

p_cgu	pointer to CGU
p_cgu_static_attrib	pointer to where CGU static attributes will be written. See description in Section 4.1.3.1.

Restrictions

CGU_New must be called first.

A CGU must match the object it is attached to. See *cgu_object_type* field in CGU Static Attributes (Section 4.1.3.1).

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

4.1.3.1 CGU Static Attributes

This structure is used in *CGU_Get_Static_Attributes* and in the list structure used in *CBME_Get_CGU_List*. All parameters except *code_number* and *cgu_index* are configured via the scanchain.

There are two structures used to describe the CGU attributes.

```
typedef struct polynomial_struct
{
        UINT16  polynomial_bit_length;
        UINT32  polynomial_value[(M_MAX_WORDS_PER_POLYNOMIAL];
```

} **M_POLYNOMIAL_TYPE**;

Polynomial values are stored in the following fashion (presume a 40 bit polynomial) within the *polynomial_value* array:

polynomial_value[0]

bits 31 - 0
$poly_{31} - poly_0$

polynomial_value[1]

Bits 31 - 8	Bit 7 - 0
Not used	$poly_{39} - poly_{32}$

```
typedef struct  cgu_static_attrib_struct
{
        UINT32                  code_number;
        UINT16                  cgu_index;
        UINT8                   cgu_name[M_MAX_LENGTH_CGU_NAME];
        UINT16                  cgu_object_type;
        UINT16                  rule;
        UINT8                   zero_insertion_enable;
        UINT8                   zero_insertion_location;
        UINT16                  num_polynomials;
        M_POLYNOMIAL_TYPE       polynomial[M_MAX_CGU_POLYNOMIALS];
        UINT32                  sequence_length;
```

} **CGU_STATIC_ATTRIB_TYPE**;

CGU Static Attributes	Description
code_number	Code number for a specified standard. This value is set when calling *CGU_New*
cgu_index	Selects which on-chip CGU to associate with this CGU object. The index references a location in the on-chip CGU list

CGU Static Attributes	Description
	generated when calling *CBME_Get_CGU_List.*
cgu_name	String name of this CGU that was assigned during configuration of the scanchain.
cgu_object_type	The type of object this CGU must be associated with. Legal values: M_SEARCHER_CGU M_UPLINK_CGU M_PDE_ANTENNA_CGU M_DOWNLINK_CGU
rule	Rule describing how the code is interpreted Legal values: TBD
zero_insertion_enable	Enables or disables zero insertion Legal values: M_TRUE or M_FALSE
zero_insertion_location	Location in the code where a zero insertion should occur. Only used if *zero_insertion_enable* is true.
num_polynomials	Number of polynomials associated with this code
polynomial	Array of polynomials for this CGU.
sequence_length	Period of the generated sequence.

4.1.4 CGU_Get_Attach_Count

Prototype

UINT16 **CGU_Get_Attach_Count**(CGU *p_cgu,
UINT16 *p_attach_count);

Description

Returns the number of objects that are attached to this CGU.

The CGU attach count is incremented every time an object attaches to it (e.g. Searcher_New (new mobile searcher), Uplink_New, PDE_Antenna_New, or Downlink_New).

The CGU attach count is decremented every time an object that was using it is freed (e.g. Searcher_Free ('new mobile' searcher), Uplink_Free, PDE_Antenna_Free, Downlink_Free)

Input Parameters

p_cgu	pointer to CGU
p_attach_count	pointer to where the number of objects attached to this CGU is written.

Restrictions

CGU_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

4.1.5 CGU_Set_User_Data

Prototype

```
UINT16 CGU_Set_User_Data(CGU    *p_cgu,
                         UINT16 index,
                         UINT16 length,
                         UINT8  *p_data);
```

Description

See Section 14.3.1 for a description of this function.

4.1.6 CGU_Get_User_Data

Prototype

```
UINT16 CGU_Get_User_Data(CGU    *p_cgu,
                         UINT16 index,
                         UINT16 length,
                         UINT8  *p_data);
```

Description

See Section 14.3.2 for a description of this function.

5 Uplink

The Uplink object is used to group Combiners (with attached Fingers) and 'existing mobile' Searchers that are common to a mobile uplink. Since the Uplink by itself does not consume any CBME resources, there is no limit to the number of Uplinks that can be attached to a CMBE.

Combiners and 'existing mobile' Searchers must be connected to an Uplink object before they can operate.

5.1 Uplink Methods

CBME_New must be called prior to any Uplink Methods. This restriction is not repeated for each function description.

5.1.1 Uplink_New

Prototype

```
UINT16 Uplink_New( CBME      *p_cbme,
                   UPLINK    *p_uplink,
                   CGU       *p_cgu);
```

Description

Allocates a new uplink.

Input Parameters

p_cbme	pointer to parent CBME
p_uplink	pointer to uplink being allocated
p_cgu	pointer to CGU (see Section 4). All Combiners and 'existing mobile' Searchers under this Uplink will use this CGU

Restrictions

p_cgu must point to an initialized CGU that is an Uplink CGU (see Section 4.1.3.1). In addition, the CGU must be associated with the same CBME as the Uplink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.2 Uplink_Free

Prototype

UINT16 **Uplink_Free**(UPLINK *p_uplink);

Description

Deallocates an uplink.

Input Parameters

p_uplink pointer to uplink

Restrictions

Uplink_New must be called first. Uplink object cannot have any objects (combiners or 'existing mobile' searchers) attached to it. These can be removed via Uplink_Remove_Combiner or Uplink_Remove_Searcher.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.3 Uplink_Set_DPCCH_Slot_Format

Prototype

UINT16 **Uplink_Set_DPCCH_Slot_Format**(UPLINK *p_uplink,
UINT16 slot_format);

Description

Sets the slot format index for the uplink. See Section 3.1.16.

Input Parameters

p_uplink	pointer to uplink
slot_format	Desired slot format
	Valid range : 0 – (*max_uplink_slot_formats** – 1)

**max_uplink_slot_formats* described in CBME Resources (see Section 3.1.12.1).

Restrictions

Uplink_New must be called first.

All Combiners currently associated with the Uplink must be in the stopped state.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.4 Uplink_Add_Combiner

Prototype

UINT16 **Uplink_Add_Combiner**(UPLINK *p_uplink, COMBINER *p_comb);

Description

Adds a combiner to an uplink.

If the uplink is in a running state (either Uplink_Start or Combiner_Start has been previously called) then the combiner being added will automatically start, and any fingers subsequently added to this combiner will automatically start.

Input Parameters

p_uplink	pointer to uplink
p_comb	pointer to combiner being added

Restrictions

Uplink_New, Uplink_Set_DPCCH_Slot_Format, Combiner_New must be called prior to this call.

The combiner must be in a stopped state (see *Combiner_Stop*, Section 11.1.8).

A maximum of four combiners can be added to an uplink.

Uplink and combiner must belong to the same CBME.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.5 Uplink_Remove_Combiner

Prototype

UINT16 **Uplink_Remove_Combiner**(COMBINER *p_comb);

Description

Removes a combiner from an uplink. Any fingers attached to the combiner will be stopped.

Input Parameters

p_comb pointer to combiner being removed

Restrictions

Uplink_Add_Combiner must be called first.

All fingers must be removed from the combiner (see Combiner_Remove_All_Fingers, Section 11.1.6)

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.6 Uplink_Add_Searcher

Prototype

UINT16 **Uplink_Add_Searcher**(UPLINK *p_uplink,
SEARCHER *p_searcher);

Description

Adds an 'existing mobile' searcher to an uplink.

If the uplink is in a running state, then the searcher will automatically start within 512 chips. If the uplink is in a stopped state, then the searcher will be in a stopped state.

Input Parameters

p_uplink	pointer to uplink
p_searcher	pointer to searcher being added

Restrictions

At least one combiner must already have been added to the uplink.

Only 'existing mobile' searchers can be added to an uplink object.

A maximum of searchers per uplink is M_MAX_SEARCHERS_PER_UPLINK .

Uplink and searcher must belong to the same CBME.

Uplink_New, Searcher_New, Searcher_Set_Existing_Mobile_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.7 Uplink_Remove_Searcher

Prototype

UINT16 **Uplink_Remove_Searcher**(SEARCHER *p_searcher);

Description

Removes an 'existing mobile' searcher from an uplink. The searcher, if running, will be stopped.

Input Parameters

p_searcher pointer to searcher being removed

Restrictions

Uplink_Add_Searcher must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.8 Uplink_Start

Prototype

```
UINT16 Uplink_Start(UPLINK    *p_uplink,
                    UINT16    frame_number,
                    UINT16    symbol_number);
```

Description

This command starts all combiners and their attached fingers as well as searchers under this uplink at the specified frame and symbol.

If no searcher is attached to the uplink this function is the equivalent of Combiner_Start().

The uplink is considered to be in a running state after this function call.

Input Parameters

p_uplink	pointer to uplink being started.
frame_number	frame number to start uplink's combiners and searchers
symbol_number	symbol number to start uplink's combiners and searchers

Restrictions

Uplink_New must be called first.

There must be at least one combiner with one finger attached to this uplink. It is not required to have a searcher attached to the uplink.

All combiners and searchers under this uplink must be in a stopped state.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.9 Uplink_Get_DPCCH_Slot_Format

Prototype

UINT16 **Uplink_Get_DPCCH_Slot_Format**(UPLINK *p_uplink,
UINT16 *p_slot_format);

Description

Gets the slot format for this uplink.

Input Parameters

p_uplink	pointer to Uplink
p_slot_format	pointer to where Uplink's slot format index will be written.

Restrictions

Uplink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.10 Uplink_Get_Num_Objects

Prototype

UINT16 **Uplink_Get_Num_Objects**(UPLINK *p_uplink,
UINT16 *p_num_searchers,
UINT16 *p_num_combiners);

Description

Returns the number of 'existing mobile' searchers and combiners that are associated with this uplink.

Input Parameters

p_uplink	pointer to uplink
p_num_searchers	pointer to where the number of 'existing mobile' searchers attached to this uplink will be written
p_num_combiners	pointer to where the number of combiners attached to this uplink will be written

Restrictions

Uplink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.11 Uplink_Get_Combiner_List

Prototype

UINT16 **Uplink_Get_Combiner_List**(UPLINK *p_uplink,
M_COMBINER_LIST_TYPE *p_comb_list);

Description

Returns a list of pointers to combiners that have been added to this uplink.

Input Parameters

p_uplink	pointer to uplink
p_comb_list	pointer to where the list of combiner pointers will be written. (see Section 5.1.11.1 for description of M_COMBINER_LIST_TYPE)

Restrictions

Uplink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.11.1 M_COMBINER_LIST_TYPE

```
typedef struct combiner_list_struct
{
  UINT16       num_combiners;
  COMBINER *p_comb[M_MAX_COMBINERS_PER_UPLINK];

} M_COMBINER_LIST_TYPE;
```

Field	Description
num_combiners	Number of combiners in the list.
p_comb	List of pointers to combiners. If *num_combiners* > 0, then the valid range is 0 to (*num_combiners* – 1).

5.1.12 Uplink_Get_Searcher_List

Prototype

UINT16 **Uplink_Get_Searcher_List**(UPLINK *p_uplink,
M_SEARCHER_LIST_TYPE *p_searcher_list);

Description

Returns a list of pointers to 'existing mobile' searchers that have been added to this uplink.

Input Parameters

p_uplink	pointer to uplink
p_searcher_list	pointer to where the list of searcher pointers will be written (see Section 5.1.12.1)

Restrictions

Uplink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

5.1.12.1 M_SEARCHER_LIST_TYPE

```
typedef struct searcher_list_struct
{
  UINT16       num_searchers;
  SEARCHER *p_searcher[M_MAX_SEARCHERS_PER_UPLINK];

} M_SEARCHER_LIST_TYPE;
```

Field	Description
num_searchers	Number of searchers in the list.
p_searcher	List of pointers to searchers. If *num_searchers* > 0, then the valid range is 0 to (*num_searchers* – 1).

5.1.13 Uplink_Get_Associated_CGU

Prototype

CGU * **Uplink_Get_Associated_CGU**(UPLINK *p_uplink,
UINT16 *p_error_code);

Description

Returns pointer to the CGU associated with this Uplink.

Input Parameters

p_uplink	pointer to uplink
p_error_code	pointer to where error code will be written.

Restrictions

Uplink_New must be called first.

Return Values

(a) valid pointer to associated CGU and *p_error_code = M_SUCCESS

or

(b) NULL and *p_error_code contains an error code (see Section 14.1)

5.1.14 Uplink_Set_User_Data

Prototype

```
UINT16  Uplink_Set_User_Data(UPLINK    *p_uplink,
                             UINT16    index,
                             UINT16    length,
                             UINT8     *p_data);
```

Description

See Section 14.3.1 for a description of this function.

5.1.15 Uplink_Get_User_Data

Prototype

```
UINT16  Uplink_Get_User_Data( UPLINK    *p_uplink,
                              UINT16    index,
                              UINT16    length,
                              UINT8     *p_data);
```

Description

See Section 14.3.2 for a description of this function.

6 Searcher

A Searcher can be configured as a 'new mobile' or 'existing mobile' Searcher. An 'existing mobile' Searcher must always be added to an Uplink. A 'new mobile' Searcher is never added to an Uplink.

6.1 Searcher Methods

CBME_New must be called prior to any Searcher Methods. This restriction is not repeated for each function description.

6.1.1 Searcher_New

Prototype

```
UINT16 Searcher_New( CBME        *p_cbme,
                     SEARCHER    *p_searcher
                     UINT16       searcher_type,
                     CGU         *p_cgu);
```

Description

Allocates a new searcher.

Input Parameters

p_cbme	pointer to parent CBME
p_searcher	pointer to searcher to be allocated
searcher_type	Type of searcher (M_NEW_MOBILE_SEARCHER or M_EXISTING_MOBILE_SEARCHER)
p_cgu	Pointer to CGU. This is only used for a 'new mobile' Searcher; see Section 4. Should be set to NULL for 'existing mobile' Searcher.

Restrictions

Total number of searchers must be less than or equal to *max_searchers* (see CBME_Get_Resource_Attributes (Section 3.1.12).

CBME_Set_Search_Periodicity must be called prior to this function call.

CBME_Set_Searcher_Energy_Scaling must be called prior to this function call.

Note that an 'existing mobile' searcher must be added to an Uplink object. A 'new mobile' Searcher is never added to an Uplink.

If the *searcher_type* is M_NEW_MOBILE_SEARCHER, then *p_cgu* must point to an initialized CGU that is a Searcher CGU (see Section 4.1.3.1). In addition, the CGU must be associated with the same CBME as the 'new mobile' Searcher.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.2 Searcher_Free

Prototype

UINT16 **Searcher_Free**(SEARCHER *p_searcher);

Description

Deallocates a searcher.

Input Parameters

p_searcher pointer to searcher

Restrictions

Searcher_New must be called first. Searcher must not be running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.3 Searcher_Set_Static_Attributes

Prototype

```
UINT16 Searcher_Set_Static_Attributes(
        SEARCHER                          *p_searcher,
        SEARCHER_NM_STATIC_ATTRIB_TYPE    *p_searcher_static_attrib);
```

Description

Sets a searcher's attributes. See Section 6.1.3.1 for details on searcher static attributes. The searcher maintains a copy of its attributes, so the attribute structure passed in may be modified after this call.

Input Parameters

p_searcher pointer to searcher

p_searcher_static_attrib pointer to searcher static attributes

Restrictions

Searcher_New must be called first.

Searcher must not be running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.3.1 Searcher Static Attributes

The 'C' structure for SEARCHER_STATIC_ATTRIB_TYPE is:

```
typedef struct  searcher_static_attrib_struct
{
        UINT16          antenna_port_num;
        UINT32          start_search_offset;
        UINT32          search_window_size;
        UINT16          pilot_gating;

} SEARCHER_STATIC_ATTRIB_TYPE;
```

Table 6-1 : Searcher Static Attributes

Searcher Static Attribute	Description
antenna_port_num	Defines the antenna data port from which the searcher will be operating. Valid Range: 0 to *max_uplink_antenna_port*
start_search_offset	Defines the initial starting point for the searcher (in chips). This is a measure of the received time of a signal relative to the base station's 0-delay reference point. Valid Range: $0 \leq (search\_start\_offset + search\_window\_size) \leq uplink\_antenna\_buffer\_size^*$ *see 3.1.12.1 for description of *uplink_antenna_buffer_size*
search_window_size	Defines how many chips should be processed by a searcher. Valid Range: see *start_search_offset*
pilot_gating	Enables or disables pilot gating. Valid Range: M_PILOT_GATING_ENABLED or M_PILOT_GATING_DISABLED

6.1.4 Searcher_Copy

Prototype

UINT16 **Searcher_Copy**(SEARCHER *p_dest_searcher,
SEARCHER *p_src_searcher);

Description

Copies the static attributes from one searcher to another.

Input Parameters

p_dest_searcher	pointer to searcher that is the destination of the static attributes
p_src_searcher	pointer to searcher that is the source of the static attributes

Restrictions

Searcher_New must be called first (for both searchers).

Searcher_Set_Static_Attributes must be called first for the source searcher.

The destination searcher cannot be running when this function is called.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.5 Searcher_Assign_DSM

Prototype

UINT16 **Searcher_Assign_DSM**(SEARCHER *p_searcher , DSM *p_dsm,)

Description

Associates a Searcher DSM with a searcher.

Input Parameters

p_searcher	pointer to searcher
p_dsm	pointer to DSM that is being added to searcher

Restrictions

Searcher_New, Searcher_DSM_New, and Searcher_DSM_Set_Static_Attributes must be called first.

Since a searcher can only have one DSM, if this function is called twice for the same searcher, the last call to this function will determine which DSM the searcher uses.

Searcher and Searcher DSM must belong to the same CBME.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.6 Searcher_Start

Prototype

UINT16 **Searcher_Start**(SEARCHER *p_searcher);

Description

Starts a searcher.

Input Parameters

p_searcher pointer to searcher to start

Restrictions

For either type of searcher ('new mobile' or 'exiting mobile', the following must be true:

a) Searcher_Assign_DSM must be called first.

b) Searcher must not be running

c) Searcher_Set_Static_Attributes must be called first.

If the searcher is created as an 'existing mobile' searcher (see *Searcher_New*), then the following must be true:

a) The searcher must be added to an uplink object before it starts (see Uplink_Add_Searcher, Section 5.1.6.

b) A combiner with at least one finger must have been previously added to the uplink and the combiner must have been started.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.7 Searcher_Stop

Prototype

UINT16 **Searcher_Stop**(SEARCHER *p_searcher);

Description

Stops a searcher.

Input Parameters

p_searcher pointer to searcher to stop.

Restrictions

Searcher_Start must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.8 Searcher_Get_Static_Attributes

Prototype

```
UINT16 Searcher_Get_Static_Attributes(
                SEARCHER                    *p_searcher,
                SEARCHER_STATIC_ATTRIB_TYPE *p_static_attrib);
```

Description

Retrieves a searcher's static attributes and copies them to the user-supplied structure. See Section 6.1.3.1 for a description of the static attributes 'C' structure and attribute definitions.

Input Parameters

p_searcher pointer to the searcher

p_static_attrib pointer to structure where attributes will be written

Restrictions

Searcher_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.9 Searcher_Get_State

Prototype

UINT16 **Searcher_Get_State**(SEARCHER *p_searcher, UINT16 *p_searcher_state);

Description

Gets a searcher state (running or stopped)

Input Parameters

p_searcher	pointer to the searcher
p_searcher_state	pointer to where searcher state is written (M_SEARCHER_RUNNING or M_SEARCHER_STOPPED)

Restrictions

Searcher_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.10 Searcher_Get_Type

Prototype

UINT16 **Searcher_Get_Type**(SEARCHER *p_searcher, UINT16 *p_type);

Description

Gets a searchers type ('new mobile' searcher or 'existing mobile' searcher). The type of searcher is set via the *searcher_type* parameter in *Searcher_New*.

Input Parameters

p_searcher	pointer to the searcher
p_type	pointer to where searcher type is written (M_NEW_MOBILE_SEARCHER or M_EXISTING_MOBILE_SEARCHER)

Restrictions

Searcher_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

6.1.11 Searcher_Get_Associated_DSM

Prototype

SEARCHER_DSM * **Searcher_Get_Associated_DSM(**
SEARCHER *p_searcher,
UINT16 *p_error_code);

Description

Returns pointer to the Searcher DSM that is associated with this searcher. A searcher can only have one associated DSM.

Input Parameters

p_searcher pointer to searcher
p_error_code pointer to where error code will be written

Restrictions

Searcher_New

Return Values

(a)Valid pointer to associated Searcher DSM
or

(b) If a NULL is returned and *p_error_code* is M_SUCCESS, then this searcher has not yet had a Searcher DSM added to it yet.
or

(c) If a NULL is returned and *p_error_code* is not M_SUCCESS, then an error occurred.

6.1.12 Searcher_Get_Associated_Uplink

Prototype

UPLINK * **Searcher_Get_Associated_Uplink**(SEARCHER *p_searcher,
UINT16 *p_error_code);

Description

Returns pointer to the Uplink that is associated with this searcher.

Input Parameters

p_searcher pointer to searcher
p_error_code pointer to where error code will be written

Restrictions

Searcher_New must be called first.

Searcher must be an 'existing mobile' searcher.

Return Values

(a)Valid pointer to associated Uplink
or

(b) If a NULL is returned and *p_error_code* is M_SUCCESS, then this searcher has not yet been added to an uplink object
or

(c) If a NULL is returned and *p_error_code* is not M_SUCCESS, then an error occurred.

6.1.13 Searcher_Get_Associated_CGU

Prototype

CGU * **Searcher_Get_Associated_CGU**(SEARCHER *p_searcher,
UINT16 *p_error_code);

Description

Returns pointer to the CGU associated with this Searcher.

Input Parameters

p_searcher	pointer to searcher
p_error_code	pointer to where error code will be written.

Restrictions

Searcher_New must be called first, and this must be a 'new mobile' Searcher.

Return Values

(a) valid pointer to associated CGU and *p_error_code = M_SUCCESS

or

(b) NULL and *p_error_code contains an error code (see Section 14.1)

6.1.14 Searcher_Set_User_Data

Prototype

```
UINT16 Searcher_Set_User_Data(SEARCHER *p_searcher,
                              UINT16    index,
                              UINT16    length,
                              UINT8     *p_data);
```

Description

See 14.3.1 for a description of this function.

6.1.15 Searcher_Get_User_Data

Prototype

```
UINT16 Searcher_Get_User_Data( SEARCHER  *p_searcher,
                               UINT16    index,
                               UINT16    length,
                               UINT8     *p_data);
```

Description

See Section 14.3.2 for a description of this function.

6.2 *Searcher Events*

This section describes the events generated by the Searcher. These events are reported via the PDE Message Queue. Refer to Section 2.3.2 to for how this queue is created and accessed.

6.2.1 Searcher Queue Messages

This section describes the format of the Searcher messages that will be sent by the VMI to the PDE Message Queue. Currently, there is only one message type, the Searcher Energy Message.

6.2.1.1 Searcher Energy Message Format

Word 1 (Header Word)

31 – 16	15 – 0
Msg Type (always SEARCHER_ENERGY_MSG)	Length

Word 2

31 – 0
Pointer to Searcher

Word 3

31 – 9	16	15 – 8	7 – 0
Not used	End Search Window Flag	Number of Results	Antenna Port Number

[if at least one energy, then Words 4 and 5)]

Word 4

31 – 16	15 – 0
Energy (for Result 1)	Offset (for Result 1)

Word 5

31 – 8	7 – 0
Not used	Phase (for Result 1)

[if two energies, then Words 6 and 7)]

Word 6

31 – 16	15 – 0
Energy (for Result 2)	Offset (for Result 2)

Word 7

31 – 8	7 – 0
Not used	Phase (for Result 2)

PDE Energy Message Field	Description
Length	Length of this message in 32-bit words. The value of this field adheres to the formula: Length = 3 + (*Number of Results* * 2) The minimum message length is 3 32-bit words (*Number of Results* = 0). There are a maximum of 2 energies that could be returned for a searcher; thus the maximum length of this message is 7 32-bit words.
Msg Type	Always equal to SEARCHER_ENERGY_MSG
Pointer to Searcher	Pointer to the searcher associated with this message. This must be cast to (SEARCHER *).
Antenna Port Number	Antenna port associated with the Searcher results returned in this message. Valid Range: 0 to *max_uplink_antenna_port*[*] [*]see CBME Resource attributes in Section 3.1.12.1
Number of Results	Number of energy results for this event. Valid Range: 0 to M_SEARCHER_MAX_RESULTS_PER_SEARCH Note: it is possible to have 0 energies in this message if the *End Search Window Flag is set.*
End Search Window Flag	Indicates the end of a search window. 0 : this message does not indicate the end of a search window 1 : this message indicates the end of a search window If this bit is set, it is possible to get this message with *Number of Results* = 0.
Offset	Energy offset (in chips)
Energy	$E_{out} = E(0) + E(1) + \ldots E(L-1)$ $E(i) = [X_i(0) + X_i(1) + \ldots X_i(N-1)]^2 + [X_q(0) + X_q(1) + \ldots X_q(N-1)]^2$ where: E_{out} : output energy N : coherent integration length in number of chips L : Post detection integration length X_i : Read part of the complex input signal X_q : Imaginary part of the complex input signal The energy output is the scaled saturated version of the sum of L energies, where each L energy is the energy of N accumulated symbols.
Phase	Energy phase

7 Searcher DSM

Each Searcher must be associated with one Dwell State Machine (DSM) which consists of a single dwell state. All searches are at ½ chip resolution.

7.1 Searcher DSM Methods

CBME_New must be called prior to any Searcher DSM methods. This restriction is not repeated for each function description.

7.1.1 Searcher_DSM_New

Prototype

UINT16 **Searcher_DSM_New**(CBME *p_cbme,
SEARCHER_DSM *p_dsm);

Description

Allocates a new Searcher DSM.

Input Parameters

p_cbme pointer to CBME
p_dsm pointer to DSM that is being allocated

Restrictions

CBME_Set_DSM_Subchip_Phase must be called prior to this function.

Number of DSMs allocated ≤ *max_dsm**

*the *max_dsm* field in CBME Resource Attributes (see Section 3.1.12.1).

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

7.1.2 Searcher_DSM_Free

Prototype UINT16 **Searcher_DSM_Free**(SEARCHER_DSM *p_dsm)
Description Deallocates a DSM.
Input Parameters **p_dsm** pointer to DSM that is being deallocated
Restrictions Searcher_DSM_New must be called first
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

7.1.3 Searcher_DSM_Set_Static_Attributes

Prototype

```
UINT16 Searcher_DSM_Set_Static_Attributes(
        SEARCHER_DSM                          *p_dsm,
        M_SEARCHER_DSM_STATIC_ATTRIB_TYPE     *p_static_attrib);
```

Description

Sets the static attributes for a Searcher DSM.

Input Parameters

p_dsm	pointer to Searcher DSM being configured
p_static_attrib	pointer to Searcher DSM static attributes. See Section 7.1.3.1 for a description of this data type.

Restrictions

Searcher_DSM_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

7.1.3.1 M_SEARCHER_DSM_STATIC_ATTRIB_TYPE

```
typedef  struct searcher_dsm_static_attrib_struct
{
        UINT16          int_length;
        UINT16          pdi_length;
        UINT16          threshold;
        UINT16          gds_shift;

} M_SEARCHER_DSM_STATIC_ATTRIB_TYPE;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>int_length</td><td>integration length (number of chips to integrate over). Only multiples of 4 are allowed.

<table>
<tr><th>int_length</th><th>Description</th></tr>
<tr><td>4</td><td>Smallest selectable integration length</td></tr>
<tr><td>8 – 255</td><td>Not allowed</td></tr>
<tr><td>256</td><td>Valid</td></tr>
<tr><td>260</td><td>Valid</td></tr>
<tr><td>264</td><td>Valid</td></tr>
<tr><td>268</td><td>Valid</td></tr>
<tr><td>.
.
.</td><td>.
.
.</td></tr>
<tr><td>2048</td><td>Largest integration length</td></tr>
</table>

Restrictions:
• If running 3GPP, multiples of 256 must be used, because the searcher works on the DPCCH, which has a spreading factor of 256.

• If the Searcher's pilot_gating (see Section 6.1.3.1) is enabled, this must be set to 256. If pilot_gating is disabled, then other values may be selected.

Valid Range: M_MIN_DSM_INTEGRATION_LENGTH to
M_MAX_DSM_INTEGRATION_LENGTH
[less the range from 8 – 255]</td></tr>
<tr><td>pdi_length</td><td>post-detection integration length

Valid Range: M_MIN_PDI_LENGTH to
M_MAX_PDI_LENGTH</td></tr>
<tr><td>threshold</td><td>pass energy threshold for state_num</td></tr>
</table>

	Valid Range: M_MIN_DSM_THRESHOLD to M_MAX_DSM_THRESHOLD
gds_shift	Generic Despread Shift. This field determines how many bits to right-shift the accumulated despread values (which are 22 bits before shifting). The most significant 14 bits of the internal 22-bit value are then used. Valid Range: 0 - 9

7.1.4 Searcher_DSM_Get_Static_Attributes

Prototype

UINT16 **Searcher_DSM_Get_Static_Attributes(**
SEARCHER_DSM *p_dsm,
M_SEARCHER_DSM_STATIC_ATTRIB_TYPE *p_static_attrib);

Description

Gets the Searcher DSM static attributes.

Input Parameters

p_dsm	pointer to DSM whose attributes are being retrieved
p_static_attrib	pointer to where the static attributes will be written

Restrictions

Searcher_DSM_Set_Static_Attributes must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

7.1.5 Searcher_DSM_Set_User_Data

Prototype

```
UINT16 Searcher_DSM_Set_User_Data(SEARCHER_DSM  *p_dsm,
                                  UINT16        index,
                                  UINT16        length,
                                  UINT8         *p_data);
```

Description

See Section 14.3.1 for a description of this function.

7.1.6 Searcher_DSM_Get_User_Data

Prototype

```
UINT16 Searcher_DSM_Get_User_Data(SEARCHER_DSM *p_dsm,
                                  UINT16        index,
                                  UINT16        length,
                                  UINT8         *p_data);
```

Description

See Section 14.3.2 for a description of this function.

8 Preamble Detection Engine

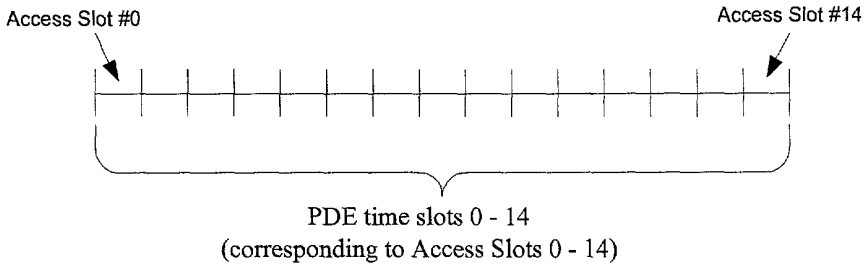
Each Preamble Detection Engine (PDE) is assigned a specific mode of operation and an access slot. A PDE can be associated with one to eight antennas depending on its mode. Similarly, the PDE search window parameters are determined by its mode. The access slots are traversed sequentially, starting with Access Slot 0.

The function *CBME_Set_PDE_Num_Slots* (Section 3.1.9) configures the total number of access slots for a specific standard, and the number of preamble detection engine time slots. For example, presume that 15 access slots are required for a CDMA standard, and the CBME supports 32 Preamble Detection Engines. One possible configuration could be:

```
CBME_Set_PDE_Num_Slots(p_cbme, 15, 1);
```

This function call configures the CBME to support one set of fifteen access slots. Under this configuration, fifteen PDE's can be defined, each corresponding to a single access slot (0 – 14). Figure 8-1 below portrays this scenario:

Figure 8-1: One Set of Access Slots (PDE Time Slots 0 – 14)

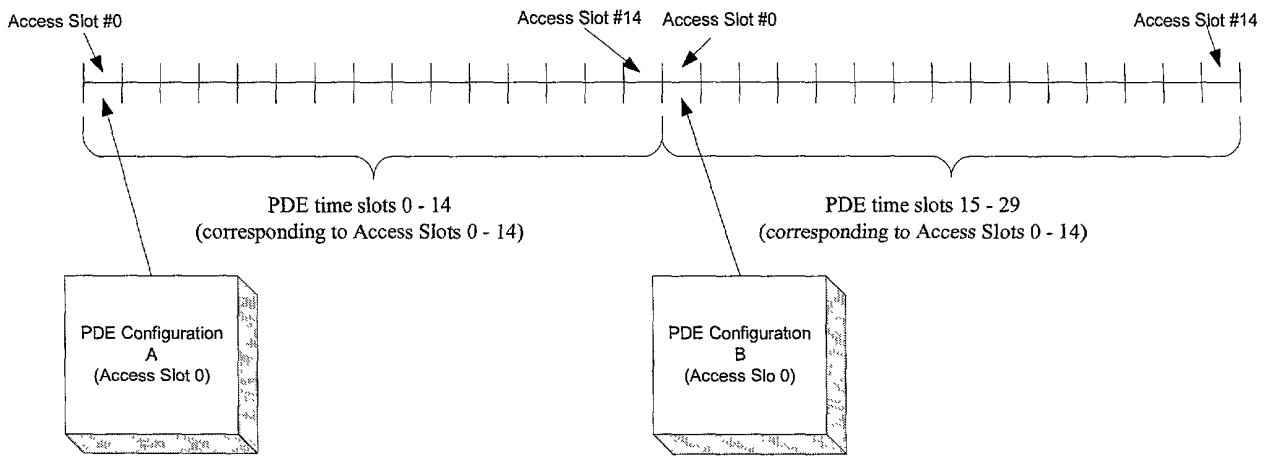


Another possible configuration could be:

```
CBME_Set_PDE_Num_Slots(p_cbme, 15, 2);
```

In this configuration, the CBME is configured to support two sets of fifteen access slots. Thus, thirty PDE's can be defined. Under this configuration, for example, one could define two unique PDE's for Access Slot 0, and they would alternate each time Access Slot 0 occurred, as shown in Figure 8-2:

Figure 8-2 : Two Sets of Access Slots (PDE Time Slots 0 – 29)



If, for example, it was desired to have Access Slot 1 be identical, then two PDE's could be configured identically and assigned to slots 1 and 16.

When the PDE's are started, they commence from time slot 0, go to the last PDE time slot, and wrap again at PDE time slot 0.

Each PDE must have one or more antennas associated with it. This is done with the PDE_Antenna object.

PDE's cannot be started and stopped individually. Either all the PDE's are all running or they are all stopped. If a configuration change is needed, then stop all PDE's, reconfigure, and start them again.

8.1 *Preamble Detection Engine Methods*

CBME_New and must be called prior to any Preamble Detection Engine methods. This restriction is not repeated for each function description

8.1.1 PDE_New

Prototype

UINT16 **PDE_New**(CBME *p_cbme,
PDE *p_pde);

Description

Allocates a new Preamble Detection Engine.

Input Parameters

p_cbme	pointer to parent CBME
p_pde	pointer to PDE that is being allocated

Restrictions

CBME_Set_PDE_Num_Slots must be called first.

The number of PDE's that can be allocated is equal to the number of time slots allocated in CBME_Set_PDE_Num_Slots.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.2 PDE_Free

Prototype

UINT16 **PDE_Free**(PDE *p_pde)

Description

Deallocates a PDE. Any antennas that have been added to this PDE are removed.

Input Parameters

p_pde pointer to PDE that is being deallocated

Restrictions

PDE_New must be called first.

PDE cannot be running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.3 PDE_Set_Static_Attributes

Prototype

```
UINT16 PDE_Set_Static_Attributes(
                PDE                         *p_pde,
                PDE_STATIC_ATTRIB_TYPE      *p_pde_static_attrib);
```

Description

Sets the Preamble Detection Engine static attributes.

Input Parameters

p_pde	pointer to PDE
p_pde_static_attrib	pointer to PDE static attributes

Restrictions

PDE_New must be called first.

Preamble Detection Engine must not be running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.3.1 Preamble Detection Engine Static Attributes Structure

The 'C' structure is:

```
typedef struct  pde_static_attrib_struct
{
        UINT16      time_slot_number;
        UINT16      mode;
        UINT16      despread_mode;
        UINT16      force_results_flag;
        UINT16      num_reports;
        UINT16      energy_scale;
        UINT16      threshold_scale;

} PDE_STATIC_ATTRIB_TYPE;
```


Table 8-1 : Preamble Detection Engine Static Attributes

Preamble Detection Engine Attribute	Description
pde_time_slot_number	PDE time slot number for this Preamble Detection Engine. If only one set of access slots has been defined (see *CBME_Set_Num_PDE_Slots*), then the PDE time slot number and access slot number are the same. If two sets of access slots have been configured, then two time slot numbers correspond to the same access slot. See explanation at start of this section (page 91). Valid Range: 0 to (number of access slots[φ] – 1) [φ]Number of access slots determined via CBME_Set_Num_PDE_Slots function.
mode	Selects the sample rate, number of antennas, number of taps, and number of hypothesis. See the Preamble Detection Engine Mode Table in the Appendix (Section 14.4). Valid Range: 0 - 71
despread_mode	Select despread mode (QPSK of OQPSK). Note that OQPSK is only supported for 2x rate (see Section 14.4). Valid Range: M_QPSK or M_OQPSK
force_results_flag	See *num_reports* field. Valid Range: M_FORCE_RESULTS_DISABLE or M_FORCE_RESULTS_ENABLE
num_reports	The number of energies per signature code to send to the microprocessor. Note that there are 16 signature codes and a maximum of 16 energies that can be reported for each antenna added to a Preamble Detection Engine. For example, if a Preamble Detection Engine had 4 antennas and was configured to report 16 energies, and all energies were above threshold, the number of energies reported would be: 16 signature codes * 16 energies * 4 antennas = 1024 energies reported If *force_results_flag* = M_FORCE_RESULTS_DISABLE, then only those energies over threshold are sent to the μP (up to a maximum of *num_reports*). If *force_results_flag* = M_FORCE_RESULTS_ENABLE, then the highest *num_reports* energies are sent independent of threshold Valid Range: M_PDE_4_ENERGIES or M_PDE_8_ENERGIES or M_PDE_12_ENERGIES or M_PDE_16_ENERGIES
energy_scale	The CBME currently supports, internally, 44-bit resolution for Preamble

<table>
<tr><th>Preamble Detection Engine Attribute</th><th>Description</th></tr>
<tr><td></td><td>Detection Engine energies. However, only 17 bits are reported to the uP. This field allows selection of which 17 bits of the 44 bits are reported to the uP. See 8.1.3.1.1 for description of how this field is used.

Valid Range:

<table>
<tr><th>Energy Bits Reported to uP</th><th>Define Value</th></tr>
<tr><td>$Energy_{16} - Energy_0$</td><td>M_PDE_ENERGY_SCALE_MSB_16 (0)</td></tr>
<tr><td>$Energy_{18} - Energy_2$</td><td>M_PDE_ENERGY_SCALE_MSB_18 (1)</td></tr>
<tr><td>$Energy_{20} - Energy_4$</td><td>M_PDE_ENERGY_SCALE_MSB_20 (2)</td></tr>
<tr><td>$Energy_{22} - Energy_6$</td><td>M_PDE_ENERGY_SCALE_MSB_22 (3)</td></tr>
<tr><td>$Energy_{24} - Energy_8$</td><td>M_PDE_ENERGY_SCALE_MSB_24 (4)</td></tr>
<tr><td>$Energy_{26} - Energy_{10}$</td><td>M_PDE_ENERGY_SCALE_MSB_26 (5)</td></tr>
<tr><td>$Energy_{28} - Energy_{12}$</td><td>M_PDE_ENERGY_SCALE_MSB_28 (6)</td></tr>
<tr><td>$Energy_{30} - Energy_{14}$</td><td>M_PDE_ENERGY_SCALE_MSB_30 (7)</td></tr>
<tr><td>$Energy_{32} - Energy_{16}$</td><td>M_PDE_ENERGY_SCALE_MSB_32 (8)</td></tr>
<tr><td>$Energy_{34} - Energy_{18}$</td><td>M_PDE_ENERGY_SCALE_MSB_34 (9)</td></tr>
<tr><td>$Energy_{36} - Energy_{20}$</td><td>M_PDE_ENERGY_SCALE_MSB_36 (10)</td></tr>
<tr><td>$Energy_{38} - Energy_{22}$</td><td>M_PDE_ENERGY_SCALE_MSB_38 (11)</td></tr>
<tr><td>$Energy_{40} - Energy_{24}$</td><td>M_PDE_ENERGY_SCALE_MSB_40 (12)</td></tr>
<tr><td>$Energy_{42} - Energy_{26}$</td><td>M_PDE_ENERGY_SCALE_MSB_42 (13)</td></tr>
<tr><td>$Energy_{43} - Energy_{27}$</td><td>M_PDE_ENERGY_SCALE_MSB_43 (14)</td></tr>
</table>
</td></tr>
<tr><td>threshold_scale</td><td>The CBME currently supports, internally, 44-bit resolution for Preamble Detection Engine energies. Each antenna associated with the PDE has a 32-bit threshold value used to compare against the energy value. This field determines which of the 44-bits of energy data the threshold is compared against. See 8.1.3.1.1 for description of how this field is used.

Valid Range:

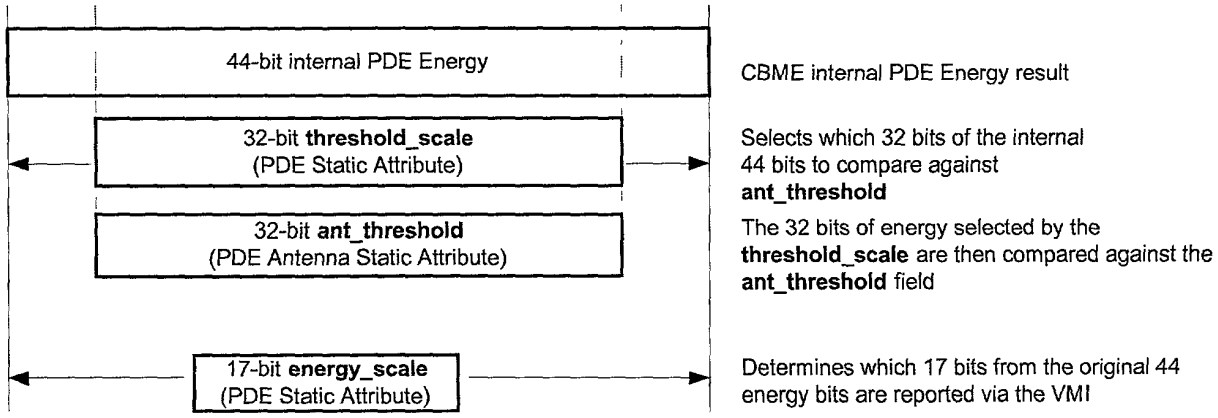
<table>
<tr><th>32-bit Theshold</th><th>Define Value</th></tr>
<tr><td>$Energy_{31} - Energy_0$</td><td>M_PDE_THRESHOLD_SCALE_MSB_31 (0)</td></tr>
<tr><td>$Energy_{32} - Energy_1$</td><td>M_PDE_THRESHOLD_SCALE_MSB_32 (1)</td></tr>
<tr><td>$Energy_{33} - Energy_2$</td><td>M_PDE_THRESHOLD_SCALE_MSB_33 (2)</td></tr>
<tr><td>$Energy_{34} - Energy_3$</td><td>M_PDE_THRESHOLD_SCALE_MSB_34 (3)</td></tr>
<tr><td>$Energy_{35} - Energy_4$</td><td>M_PDE_THRESHOLD_SCALE_MSB_35 (4)</td></tr>
<tr><td>$Energy_{36} - Energy_5$</td><td>M_PDE_THRESHOLD_SCALE_MSB_36 (5)</td></tr>
<tr><td>$Energy_{37} - Energy_6$</td><td>M_PDE_THRESHOLD_SCALE_MSB_37 (6)</td></tr>
<tr><td>$Energy_{38} - Energy_7$</td><td>M_PDE_THRESHOLD_SCALE_MSB_38 (7)</td></tr>
<tr><td>$Energy_{39} - Energy_8$</td><td>M_PDE_THRESHOLD_SCALE_MSB_39 (8)</td></tr>
<tr><td>$Energy_{40} - Energy_9$</td><td>M_PDE_THRESHOLD_SCALE_MSB_40 (9)</td></tr>
<tr><td>$Energy_{41} - Energy_{10}$</td><td>M_PDE_THRESHOLD_SCALE_MSB_41 (10)</td></tr>
<tr><td>$Energy_{42} - Energy_{11}$</td><td>M_PDE_THRESHOLD_SCALE_MSB_42 (11)</td></tr>
<tr><td>$Energy_{43} - Energy_{12}$</td><td>M_PDE_THRESHOLD_SCALE_MSB_43 (12)</td></tr>
</table>
</td></tr>
</table>

Preamble Detection Engine Attribute	Description
	$Energy_{43} - Energy_{12}$ | M_PDE_THRESHOLD_SCALE_MSB_43 (12) See *ant_threshold* field in PDE Antenna Static attributes (Section 9.1.3.1).

8.1.3.1.1 PDE Energy Scaling and Reporting

The scaling and reporting of the Preamble Detection Engine energies are controlled by three fields – *threshold_scale* and *energy_scale* in the PDE Static Attributes, and *ant_threshold* in the PDE Antenna Static Attributes. Figure 8-3 shows the interrelationship between these three fields.

Figure 8-3 : Energy Scaling and Reporting



8.1.4 PDE_Add_Antenna

Prototype

UINT16 **PDE_Add_Antenna** (PDE *p_pde,
PDE_ANT *p_pde_ant);

Description

Adds one antenna to a Preamble Detection Engine. The same antenna can be added to multiple Preamble Detection Engines.

Input Parameters

p_pde pointer to the preamble detection engine
p_pde_ant pointer to antenna being added

Restrictions

PDE_Set_Static_Attributes must be called.

PDE_Antenna_Set_Static_Attributes must be called.

PDE must not be running.

The number of antennas added to a PDE cannot exceed the number of antennas supported by the mode selected in PDE_Set_Static_Attributes.

The PDE and the PDE Antenna must belong to the same CBME.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.5 PDE_Remove_Antenna

Prototype

UINT16 **PDE_Remove_Antenna** (PDE *p_pde,
PDE_ANT *p_pde_ant);

Description

Removes one antenna from a Preamble Detection Engine.

Input Parameters

p_pde pointer to preamble detection engine
p_pde_ant pointer to antenna being removed

Restrictions

Antenna must have been added to this Preamble Detection Engine.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.6 PDE_Start_All

Prototype

UINT16 **PDE_Start_All**(CBME *p_cbme);

Description

Starts all the Preamble Detection Engines that are properly configured. The list of running PDEs can be obtained from PDE_Get_Active_List.

Input Parameters

p_cbme pointer to CBME

Restrictions

Only PDE's that meet the following criteria will be started:

a) PDE_New called
b) PDE_Set_Static_Attributes called
c) PDE_Add_Antenna called for each antenna required per the mode of the Preamble Detection Engine (mode is set in PDE_Set_Static_Attributes). For example, if the mode requires 6 antennas, then 6 antennas must have been added to this Preamble Detection Engine.

PDE's cannot be running when this function is called.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.7 PDE_Stop_All

Prototype

UINT16 **PDE_Stop_All**(CBME *p_cbme);

Description

Stops all Preamble Detection Engines that are running.

Input Parameters

p_cbme pointer to CBME

Restrictions

PDE_Start_All must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.8 PDE_Get_Active_List

Prototype

UINT16 **PDE_Get_Active_List**(CBME *p_cbme,
PDE_ACTIVE_LIST_TYPE *p_pde_list);

Description

Gets list of PDE's that are currently running

Input Parameters

p_cbme pointer to CBME

p_pde_list pointer to where list of PDEs will be written (see Section 8.1.8.1).

Restrictions

PDE_Start_All must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.8.1 PDE_ACTIVE_LIST_TYPE

```
typedef struct pde_active_list_type
{
    UINT16      num_pde;
    PDE         *p_pde[M_MAX_PDE];

} PDE_ACTIVE_LIST_TYPE;
```

Field	Description
num_pde	Number of PDEs in the list.
p_pde	Array of pointers to PDEs. If *num_pde* > 0, then the valid range is 0 to (*num_pde* – 1).

8.1.9 PDE_Get_Static_Attributes

Prototype

UINT16 **PDE_Get_Static_Attributes(**
PDE *p_pde,
PDE_STATIC_ATTRIB_TYPE *p_pde_static_attrib);

Description

Gets the Preamble Detection Engine static attributes.

Input Parameters

p_pde	pointer to PDE
p_pde_static_attrib	pointer to where PDE static attributes will be written

Restrictions

PDE_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.10 PDE_Get_State

Prototype

UINT16 **PDE_Get_State**(CBME *p_cbme, UINT16 *p_state);

Description

Returns the state of the PDE's (running or stopped).

Input Parameters

p_cbme	pointer to CBME
p_state	pointer to where state will be written
	Return values: M_PDE_RUNNING or M_PDE_STOPPED
	Note: All Preamble Detection Engines are either running or stopped.

Restrictions

PDE_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.11 PDE_Get_Antenna_List

Prototype

UINT16 **PDE_Get_Antenna_List**(PDE *p_pde,
M_PDE_ANT_LIST_TYPE *p_pde_ant_list);

Description

Returns a list of pointers to PDE antennas that have been added to this PDE (see Section 8.1.11.1 for a description of M_PDE_ANT_LIST_TYPE).

Input Parameters

p_pde	pointer to PDE
p_pde_ant_list	pointer to where antenna list will be written

Restrictions

PDE_Set_Static_Attributes must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

8.1.11.1 PDE_ANT_LIST_TYPE

```
typedef struct pde_ant_list_struct
{
  UINT16      num_pde_ant;
  PDE_ANT     *p_pde_ant[M_MAX_ANT_PER_PDE];

} M_PDE_ANT_LIST_TYPE;
```

Field	Description
num_pde_ant	Number of PDE antennas in the list.
p_pde_ant	Array of pointers to PDE antennas. If *num_pde_ant* > 0, then the valid range is 0 to (*num_pde_ant* – 1).

8.1.12 PDE_Set_User_Data

Prototype

```
UINT16 PDE_Set_User_Data(PDE     *p_pde,
                         UINT16  index,
                         UINT16  length,
                         UINT8   *p_data);
```

Description

See Section 14.3.1 for a description of this function.

8.1.13 PDE_Get_User_Data

Prototype

```
UINT16 PDE_Get_User_Data(PDE     *p_pde,
                         UINT16  index,
                         UINT16  length,
                         UINT8   *p_data);
```

Description

See Section 14.3.2 for a description of this function.

8.2 Preamble Detection Engine Events

This section describes the events generated by the PDE. These events are reported via the PDE Message Queue. Refer to Section 2.3.2 to for how this queue is created and accessed.

8.2.1 PDE Queue Messages

This section describes the format of the PDE messages that will be sent by the VMI to the PDE Message Queue. Currently, there is only one message type, the PDE Energy Message.

8.2.1.1 PDE Energy Message Format

Word 1 (Header Word)

31 – 16	15 - 0
Msg Type (always PDE_ENERGY_MSG)	Length

Word 2

31 – 0
Pointer to PDE

Word 3

31 – 24	23 – 16	15 – 8	7 – 0
PDE Time Slot Number	Access Slot Number	Signature Code	Antenna Port Number

Word 4

31 – 0
Number of Results

Word 5

31 – 0
Energy (for Result 1)

Word 6

31 – 18	17	16	15 – 0
Not used	Threshold Flag (for Result 1)	Phase (for Result 1)	Offset (for Result 1)

. **(if more than one energy)**

[Word (N - 1)]

31 – 0
Energy (for last result)

Word N

31 – 18	17	16	15 – 0
Not used	Threshold Flag (for last result)	Phase (for last result)	Offset (for last result)

PDE Energy Message Field	Description
Length	Length of this message in 32-bit words. The value of this field adheres to the formula: Length = 4 + (*Number of Results* * 2) For example, if *Number of Results* = 5, this message would have a total length of 4 + (5 * 2), or 14 32-bit words. The minimum message length is 6 32-bit words (*Number of Results* = 1). There are a maximum of 16 energies that could be returned for a signature code; thus the maximum length of this message is 36 32-bit words.
Msg Type	Always equal to PDE_ENERGY_MSG
Pointer to PDE	This is the pointer to the PDE associated with this message. It must be cast to (PDE *).
Antenna Port Number	Antenna port associated with the PDE energies returned in this message. Valid Range: M_MIN_PDE_ANT_NUM to M_MAX_PDE_ANT_NUM
Signature Code	The signature code associated with the energies returned in this message. Signature code for this result Valid Range: M_MIN_SIGNATURE_CODE to M_MAX_SIGNATURE_CODE
Access Slot Number	Access slot number for this result Valid Range: 0 to (number of access slots* – 1) *Number of access slots defined in CBME_Set_PDE_Num_Slots
PDE Time Slot Number	PDE time slot number associated with this result Valid Range: 0 to (number of time slots* – 1) *Number of PDE time slots defined in CBME_Set_PDE_Num_Slots.
Number of Results	Number of energy results for this signature code. Valid Range: 1 to M_PDE_MAX_RESULTS_PER_SEARCH
Energy	PDE scaled energy magnitude.
Offset	PDE offset (in chips)
Phase	The phase associated with this energy result. Only applicable if 2X oversampling. 0 : first of two samples 1 : second of two samples If 1X oversampling, phase selection is based on I-data selection control bit.
Threshold Flag	This flag indicates status about the this energy compared to threshold. 0 : energy may or may not be greater than threshold. This could be the

	case if the *force_results_flag* field in PDE static attributes is set true (see Section 8.1.3.1). 1 : energy is greater than threshold

9 Preamble Detection Engine (PDE) Antenna

The PDE Antenna objects are 'attached' to PDE objects.

9.1 PDE Antenna Methods

CBME_New must be called prior to any PDE Methods. This restriction is not repeated for each function description.

9.1.1 PDE_Antenna_New

Prototype UINT16 **PDE_Antenna_New**(CBME *p_cbme, PDE_ANT *p_pde_ant, CGU *p_cgu);
Description Allocates a new PDE antenna object.
Input Parameters **p_cbme** pointer to CBME. **p_pde_ant** pointer to PDE antenna that is being allocated **p_cgu** pointer to CGU for this antenna (see Section 4)
Restrictions p_cgu must point to a CGU that has had CGU_New called CGU must be a PDE CGU CGU and PDE Antenna must belong to the same CBME
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

9.1.2 PDE_Antenna_Free

Prototype

UINT16 **PDE_Antenna_Free**(PDE_ANT *p_pde_ant)

Description

Deallocates a PDE antenna object.

Input Parameters

p_pde_ant pointer to PDE antenna

Restrictions

PDE_Antenna_New must be called first

PDE_Antenna must not be attached to any PDE's

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

9.1.3 PDE_Antenna_Set_Static_Attributes

Prototype

UINT16 **PDE_Antenna_Set_Static_Attributes**(
PDE_ANT *p_pde_ant,
PDE_ANT_STATIC_ATTRIB_TYPE *p_pde_ant_static_attrib);

Description

Sets Preamble Detection Engine Antenna static attributes.

Input Parameters

p_pde_ant pointer to PDE antenna
p_pde_static_attrib pointer to PDE antenna static attributes

Restrictions

PDE's must be stopped

PDE_Antenna_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

9.1.3.1 Preamble Detection Engine Antenna Static Attributes Structure

The 'C' structure is:

```
typedef struct  pde_ant_static_attrib_struct
{
        UINT16          ant_port_number;
        UINT16          ant_phase_select;
        UINT32          ant_threshold;

} PDE_ANT_STATIC_ATTRIB_TYPE;
```

Table 9-1 : Preamble Detection Engine Antenna Static Attributes

PDE Attribute	Description
ant_port_number	CBME antenna number associated with this object. Valid Range: 0 to M_MAX_PDE_ANT_NUM
ant_phase_select	This field only applicable when the PDE Mode (see *mode* field in Section 8.1.3.1) corresponds to a 1X sample rate. The legal values are: **Value (define)** / **Description** M_PDE_ANT_I_FIRST_Q_FIRST — I : 1st of 2x over-sampled data; Q : 1st of 2x over-sampled data M_PDE_ANT_I_FIRST_Q_SECOND — I : 1st of 2x over-sampled data; Q : 2nd of 2x over-sampled data M_PDE_ANT_I_SECOND_Q_FIRST — I : 2nd of 2x over-sampled data; Q : 1st of 2x over-sampled data M_PDE_ANT_I_SECOND_Q_SECOND — I : 2nd of 2x over-sampled data; Q : 2nd of 2x over-sampled data
ant_threshold	Antenna threshold; used to compare against received energies. Valid Range: $0 - 2^{31}$ (see *theshold_scale* in PDE Static Attributes (Section 8.1.3.1). See 8.1.3.1.1 for description of how this field is used.

9.1.4 PDE_Antenna_Get_Static_Attributes

Prototype

UINT16 **PDE_Antenna_Get_Static_Attributes(**
PDE_ANT *p_pde_ant
PDE_ANT_STATIC_ATTRIB_TYPE *p_pde_ant_static_attrib);

Description

Gets Preamble Detection Engine Antenna static attributes.

Input Parameters

p_pde_ant	pointer to PDE antenna
p_pde_static_attrib	pointer to where PDE antenna static attributes will be written.

Restrictions

PDE_Antenna_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

9.1.5 PDE_Antenna_Get_Associated_CGU

Prototype

CGU * **PDE_Ant_Get_Associated_CGU**(PDE_ANT *p_pde_ant,
UINT16 *p_error_code);

Description

Returns pointer to the CGU associated with this Preamble Detection Engine Antenna.

Input Parameters

p_pde_ant	pointer to PDE Antenna
p_error_code	pointer to where error code will be written.

Restrictions

PDE_Antenna_New must be called first

Return Values

(a) valid pointer to associated CGU and *p_error_code = M_SUCCESS

or

(b) NULL and *p_error_code contains an error code (see Section 14.1)

9.1.6 PDE_Antenna_Set_User_Data

Prototype

```
UINT16 PDE_Antenna_Set_User_Data(PDE_ANT    *p_pde_ant,
                                 UINT16     index,
                                 UINT16     length,
                                 UINT8      *p_data);
```

Description

See Section 14.3.1 for a description of this function.

9.1.7 PDE_Antenna_Get_User_Data

Prototype

```
UINT16 PDE_Get_Antenna_User_Data(PDE_ANT    *p_pde,
                                 UINT16     index,
                                 UINT16     length,
                                 UINT8      *p_data);
```

Description

See Section 14.3.2 for a description of this function.

10 Finger

Tracking fingers are allocated and then added to a combiner. A finger cannot run until it has been added to a combiner.

10.1 Finger Methods

CBME_New must be called prior to any Finger methods. This restriction is not repeated for each function description.

10.1.1 Finger_New

Prototype UINT16 **Finger_New**(CBME *p_cbme, FINGER *p_finger);
Description Allocates a new finger.
Input Parameters **p_cbme** pointer to parent CBME **p_finger** pointer to the finger being allocated
Restrictions Total number of fingers allocated must be ≤ *max_fingers* field in CBME Resource Attributes (see Section 3.1.12.1 CBME_Set_Mobile_Resources must be called first.
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.2 Finger_Free

Prototype

UINT16 **Finger_Free**(FINGER *p_finger);

Description

Deallocates a finger.

Input Parameters

p_finger pointer to the finger being deallocated

Restrictions

Finger_New must be called first.

The finger must not be attached to a combiner.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.3 Finger_Set_Static_Attributes

Prototype

UINT16 **Finger_Set_Static_Attributes**(
FINGER *p_finger,
FINGER_STATIC_ATTRIB_TYPE *p_finger_static_attrib);

Description

Sets the finger static attributes. See Section 10.1.3.1 for details on the static attributes. The finger maintains a copy of its attributes, so the static attribute structure passed in may be modified after this call.

Input Parameters

p_finger	pointer to finger
p_finger_static_attrib	pointer to finger static attributes

Restrictions

Finger_New must be called first.

This call must be made before the finger is running, and may not be called after the finger is running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.3.1 Finger Static Attributes Structure

The 'C' structure is:

```
typedef struct  finger_static_attrib_struct
{
        UINT16          antenna_port_num;
        UINT16          time_delay_offset;
        UINT16          fractional_offset;

} FINGER_STATIC_ATTRIB_TYPE;
```

Table 10-1 : Finger Static Attributes

Finger Attribute	Description
antenna_port_num	Defines the antenna data port from which the finger will be operating. Valid Range: 0 to *max_uplink_antenna_port*
time_delay_offset	Defines the initial starting point for the finger (in chips). This is a measure of the received time of a signal relative to the base station's 0-delay reference point. Valid Range: $0 \leq$ *time_delay_offset* $\leq$ *uplink_antenna_buffer_size** *see 3.1.12.1 for description of *uplink_antenna_buffer_size*
fractional_offset	Initial fractional delay (in chips). This selects the subphase out of the interpolation filter. Valid Range: **Fractional Offset Field** **Description** / **Define** 0 Chips / M_0_EIGHTHS_CHIPS 1/8 chip / M_1_EIGHTHS_CHIPS 2/8 chip / M_2_EIGHTHS_CHIPS 3/8 chip / M_3_EIGHTHS_CHIPS 4/8 chip / M_4_EIGHTHS_CHIPS 5/8 chip / M_4_EIGHTHS_CHIPS 6/8 chip / M_6_EIGHTHS_CHIPS 7/8 chip / M_7_EIGHTHS_CHIPS

10.1.4 Finger_Start

Prototype
UINT16 **Finger_Start**(FINGER *p_finger);

Description
Starts a finger.

Input Parameters
p_finger pointer to finger to start

Restrictions
Finger_Set_Static_Attributes, Combiner_Add_Finger must be called first.

The finger must have been added to a combiner, and the combiner must be running.

Return Values
M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.5 Finger_Stop

Prototype
UINT16 **Finger_Stop**(FINGER *p_finger);

Description
Stops a finger.

Input Parameters
p_finger pointer to finger to stop

Restrictions
The finger must be running (Finger_Start or Combiner_Start)

Return Values
M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.6 Finger_Copy

Prototype

UINT16 **Finger_Copy**(FINGER *p_dest_finger, FINGER *p_src_finger);

Description

Copies the static attributes from one finger to another.

Input Parameters

p_dest_finger	pointer to finger that is the destination of the static attributes
p_src_finger	pointer to finger that is the source of the static attributes

Restrictions

Finger_New, Finger_Set_Static_Attributes must be called first.

The destination finger cannot be running when this function is called.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.7 Finger_Request_Offset

Prototype

UINT16 **Finger_Request_Offset**(FINGER *p_finger);

Description

Requests a read of the fingers offset. This function does not return the offset. Instead, the response will be sent via the Combiner DSP Message Queue (see Section 11.2.1.2).

This function will be eliminated for V1.1 of the CBME; it will be replaced by a message sent via Combiner_DSP_Send_Msg (see Section 11.1.9).

Input Parameters

p_finger	pointer to the finger

Restrictions

Combiner_Start must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.8 Finger_Get_ID

Prototype

UINT16 **Finger_Get_ID**(FINGER *p_finger, UINT16 *p_finger_id);

Description

Returns the finger ID of a finger that has been added to a combiner. This ID is used to identify fingers returning their energies via the Combiner_DSP_Get_Msg. Finger IDs can also be obtained via the Combiner_Get_Finger_List function.

Input Parameters

p_finger	pointer to finger
p_finger_id	pointer to where finger ID will be written.

Restrictions

The finger must be added to a Combiner.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.9 Finger_Get_Static_Attributes

Prototype

```
UINT16 Finger_Get_Static_Attributes(
                    FINGER                  *p_finger,
                    FINGER_STATIC_ATTRIB_TYPE *p_static_attrib);
```

Description

Retrieves the finger static attributes and copies them to the user-supplied structure. See Section 10.1.3.1 for structure definition.

Input Parameters

p_finger	pointer to the finger
p_static_attrib	pointer to table where attributes will be copied.

Restrictions

Finger_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

10.1.10 Finger_Set_User_Data

Prototype

```
UINT16 Finger_Set_User_Data( FINGER   *p_finger,
                             UINT16   index,
                             UINT16   length,
                             UINT8    *p_data);
```

Description

See Section 14.3.1 for a description of this function.

10.1.11 Finger_Get_User_Data

Prototype

```
UINT16 Finger_Get_User_Data( FINGER   *p_finger,
                             UINT16   index,
                             UINT16   length,
                             UINT8    *p_data);
```

Description

See Section 14.3.2 for a description of this function.

11 Combiner

The Combiner object groups fingers that are being combined. The Combiner must always be added to an Uplink object.

11.1 Combiner Methods

CBME_New must be called prior to any Combiner methods. This restriction is not repeated for each function description.

11.1.1 Combiner_New

Prototype UINT16 **Combiner_New**(CBME *p_cbme, COMBINER *p_comb);
Description Allocates a new combiner.
Input Parameters **p_cbme** pointer to parent CBME **p_comb** pointer to combiner to be allocated
Restrictions CBME_Set_Mobile_Resources must be called first. Total number of combiners allocated must be ≤ *max_combiners* field in CBME Resource Attributes (see Section 3.1.12.1)
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.2 Combiner_Free

Prototype

UINT16 **Combiner_Free**(COMBINER *p_comb);

Description

Deallocates a combiner.

Input Parameters

p_comb pointer to the combiner to be deallocated

Restrictions

Combiner_New must be called first.

Combiner must be stopped.

All fingers must be removed from combiner.

Combiner cannot be attached to an uplink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.3 Combiner_Set_Static_Attributes

Prototype

UINT16 **Combiner_Set_Static_Attributes(**
COMBINER *p_comb,
COMBINER_STATIC_ATTRIB_TYPE *p_static_attrib);

Description

Sets the static attributes for a combiner.

Input Parameters

p_comb	pointer to the combiner
p_static_attrib	pointer to combiner static attributes. See Section

Restrictions

Combiner_New must be called first.

Combiner must be stopped.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.3.1 COMBINER_STATIC_ATTRIB_TYPE

The 'C' structure is:

```
typedef combiner_static_attrib_struct
{
        UINT16      frame_offset;
        UINT16      spreading_factor[MAX_RX_CHAN_CODES];
        UINT16      channelization_number[MAX_RX_CHAN_CODES];

} COMBINER_STATIC_ATTRIB_TYPE;
```

Table 11-1 : Combiner Static Attributes

Combiner Attribute	Description
frame_offset	Initial frame offset in 256-chip intervals (e.g. *frame_offset* = 2 corresponds to an initial frame offset of 512 chips). Valid Range: M_UPLINK_MIN_FRAME_OFFSET to M_UPLINK_MAX_FRAME_OFFSET
spreading_factor	Array of initial spreading factors corresponding to channels. Index 0 corresponds to Channel 0, index 1 corresponds to Channel 1, and so on. The number of available channels for each finger Each *spreading_factor* has a corresponding (matching index) *channelization_number*. For example, *spreading_factor*[2] corresponds to *channelization_number*[2]. When the finger is running, the spreading factor can be changed via Combiner_DSP_Send_Msg (see Section 11.1.9). Valid range: M_UPLINK_CHANNEL_NOT_USED or M_SPREADING_FACTOR_4 or M_SPREADING_FACTOR_8 or M_SPREADING_FACTOR_16 or M_SPREADING_FACTOR_32 or M_SPREADING_FACTOR_64 or M_SPREADING_FACTOR_128 or M_SPREADING_FACTOR_256
channelization_number	Array of initial channelization numbers corresponding to the *spreading_factor* array. When the finger is running, the spreading factor can be changed via Combiner_DSP_Send_Msg (see Section 11.1.9). Valid Range: The *channelization_number* must always be less than the *spreading factor*. For example, if the *spreading_factor* =

	M_SPREADING_FACTOR_16, then the *channelization_number* must be 15 or less. Ignored if *spreading_factor* = M_UPLINK_CHANNEL_NOT_USED.

11.1.4 Combiner_Add_Finger

Prototype

UINT16 **Combiner_Add_Finger**(COMBINER *p_comb,
FINGER *p_finger);

Description

Adds a finger to a combiner.

If the combiner has been started (Combiner_Start or Uplink_Start), then any finger added to it will automatically start within 512 chips. If a combiner is in a stopped state, then any finger added to it will be in a stopped state.

Input Parameters

p_comb	pointer to the combiner
p_finger	pointer to finger to be added to combiner

Restrictions

Combiner_New must be called first.

Combiner_Set_Static_Attributes must be called first.

Finger_Set_Static_Attributes must be called first.

Combiner and finger must belong to the same CBME.

The combiner must be added to an uplink object before any fingers are added to it.

A maximum of M_MAX_FINGERS_PER_COMBINER fingers may be added to a combiner.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.5 Combiner_Remove_Finger

Prototype

UINT16 **Combiner_Remove_Finger**(COMBINER *p_comb,
FINGER *p_finger);

Description

Removes a finger from a combiner. If the finger is running, it will be stopped first.

Input Parameters

p_comb pointer to combiner
p_finger pointer to finger to be removed from combiner

Restrictions

The finger must have been added to this combiner via Combiner_Add_Finger

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.6 Combiner_Remove_All_Fingers

Prototype

UINT16 **Combiner_Remove_All_Fingers**(COMBINER *p_comb);

Description

Removes all fingers from a combiner. If finger is running, it will be stopped before removal.

Input Parameters

p_comb pointer to combiner

Restrictions

Combiner_Add_Finger must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.7 Combiner_Start

Prototype

UINT16 **Combiner_Start**(COMBINER *p_comb,
UINT16 frame_number,
UINT16 symbol_number);

Description

Start all fingers associated with the combiner at the specified frame and symbol. If it is desired to start searchers and fingers at the same frame/symbol, then use Uplink_Start (see Section 5.1.8).

The uplink is considered to be in a running state after this function call.

Input Parameters

p_comb	pointer to the combiner to start
frame_number	Frame number to start all fingers added to this combiner
symbol_number	symbol number to start all fingers added to this combiner

Restrictions

Combiner must be in a stopped state.

A combiner must have a least one finger added to it before it can start.

A combiner cannot be started until it is added to an uplink object (see Uplink_Add_Combiner, Section 5.1.3).

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.8 Combiner_Stop

Prototype

UINT16 **Combiner_Stop**(COMBINER *p_comb);

Description

Stops the Combiner and all fingers associated with the combiner.

Input Parameters

p_comb	pointer to the combiner to stop

Restrictions

Combiner_Start must be called first

Return Values
M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.9 Combiner_DSP_Send_Msg

Prototype

```
UINT16 Combiner_DSP_Send_Msg(
            COMBINER                *p_comb,
            M_COMB_DSP_MSG_TYPE     *p_comb_dsp_msg);
```

Description
This function sends a message to the Combiner's DSP. If this function generates a response, it will come via the Combiner DSP Message Queue (see Section 11.2.1.1).

Input Parameters

p_comb	pointer to combiner
p_comb_dsp_msg	pointer to msg being sent to Combiner DSP. See Section 11.1.9.1 for description of data type.

Restrictions
Combiner_Set_Static_Attribiutes must be called first.

Return Values
M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.9.1 M_COMB_DSP_MSG_TYPE

```
typedef struct combiner_dsp_msg_struct
{
  UINT16  msg_word[MAX_WORDS_PER_DSP_MSG];  /* 2 16-bit words for CBME V1.05 */

} M_COMBINER_DSP_MSG_TYPE;
```


11.1.10 Combiner_Get_Static_Attributes

Prototype

UINT16 **Combiner_Get_Static_Attributes(**
COMBINER *p_comb,
COMBINER_STATIC_ATTRIB_TYPE *p_static_attrib);

Description

Retrieves the static attributes that were set with Combiner_Set_Static_Attributes (see Section 11.1.3).

Input Parameters

p_comb	pointer to combiner
p_static_attrib	structure where static attributes will be written. Data type described in Section 11.1.3.1).

Restrictions

Combiner_Set_Static_Attributes must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.11 Combiner_Get_Finger_List

Prototype

UINT16 **Combiner_Get_Finger_List**(COMBINER *p_comb,
M_FINGER_LIST_TYPE *p_finger_list);

Description

Fills in the list with fingers that have been added to this combiner. The list is filled in starting from index 0 to (num fingers – 1). Unused entries are nulled out.

The position of fingers in the list is not static. In other words, the index of a specific finger in the list may vary from call to call.

Input Parameters

p_comb	pointer to combiner
p_finger_list	structure where list of finger pointers will be written (see Section 11.1.11.1 for the definition of the M_FINGER_LIST_TYPE)

Restrictions

Combiner_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.11.1 M_FINGER_LIST_TYPE

```
typedef struct finger_list_struct
{
  UINT16      num_fingers;
  FINGER      *p_finger[M_MAX_FINGERS_PER_COMBINER];
  UINT16      finger_ID[M_MAX_FINGERS_PER_COMBINER];

} M_FINGER_LIST_TYPE;
```

Field	Description
num_fingers	Number of fingers in the list
p_finger	List of finger pointers. If *num_fingers* > 0, then valid range is 0 to (*num_fingers* – 1).
finger_ID	Corresponding list of finger IDs. If *num_fingers* > 0, then valid range is 0 to (*num_fingers* – 1).

11.1.12 Combiner_Get_Associated_Uplink

Prototype

UPLINK * **Combiner_Get_Associated_Uplink**(COMBINER *p_comb,
UINT16 *p_error_code);

Description

Returns the uplink associated with this combiner.

Input Parameters

p_comb pointer to combiner
p_error_code pointer to where error code is written.

Restrictions

Combiner_New must be called first.

Return Values

(a)Valid pointer to associated Uplink
or

(b) If a NULL is returned and *p_error_code* is M_SUCCESS, then the combiner has not yet been added to an uplink object
or

(c) If a NULL is returned and *p_error_code* is not M_SUCCESS, then an error occurred (see Section 14.1)

11.1.13 Combiner_Get_State

Prototype

UINT16 **Combiner_Get_State**(FINGER *p_comb, UINT16 *p_comb_state);

Description

Gets a combiner state (running or stopped)

Input Parameters

p_comb	pointer to the combiner
p_comb_state	Pointer to where combiner state is written (M_COMBINER_RUNNING or M_COMBINER_STOPPED)

Restrictions

Combiner_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.14 Combiner_Get_Num_Fingers

Prototype

UINT16 **Combiner_Get_Num_Fingers**(COMBINER *p_comb,
UINT16 *p_num_fingers);

Description

Returns the number of fingers that have been added to the combiner.

Input Parameters

p_comb	pointer to the combiner
p_num_fingers	pointer to where the number of fingers in the combiner will be written

Restrictions

Combiner_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.15 Combiner_Get_ID

Prototype

UINT16 **Combiner_Get_ID**(COMBINER *p_comb,
UINT16 *p_comb_id);

Description

Returns the ID associated with this combiner. This ID is encoded on the CBME control bus along with mobile data.

Input Parameters

p_comb	pointer to the combiner
p_comb_id	pointer to where the combiner ID will be written

Restrictions

Combiner_New must be called first

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

11.1.16 Combiner_Set_User_Data

Prototype

UINT16 **Combiner_Set_User_Data**(COMBINER *p_comb,
UINT16 index,
UINT16 length,
UINT8 *p_data);

Description

See Section 14.3.1 for description of this function.

11.1.17 Combiner_Get_User_Data

Prototype

UINT16 **Combiner_Get_User_Data**(COMBINER *p_comb,
UINT16 index,
UINT16 length,
UINT8 *p_data);

Description

See Section 14.3.2 for description of this function.

11.2 Combiner DSP Events

This section describes the event messages generated by the Combiner DSP. These events are reported via the Combiner DSP Message Queue. Refer to Section 2.3.2 to for how this queue is created and accessed.

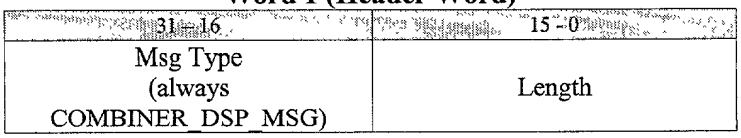
Combiner DSP Message Types	Description
Combiner DSP Message	This message will contain all the different messages sent from the Combiner DSP associated with a specific combiner. For CBME V1.1, this will be the only message type supported. For CBME V1.05, however, the *Finger Offset Message* is needed. The Combiner DSP software can be written by Morphics or by the user. Typically, the Combiner DSP will send finger energies and other data to the Combiner message queue. If the Combiner DSP software is written by Morphics, a separate document will describe
Finger Offset Message	For CBME V1.05, this needs to be a separate message that will use the Combiner DSP Message Queue (even though its not really a message from the Combiner DSP). For CBME V1.1, this information will come from the Combiner DSP, and will be incorporated into the Combiner DSP's message set.

11.2.1.1 Combiner DSP Message Format

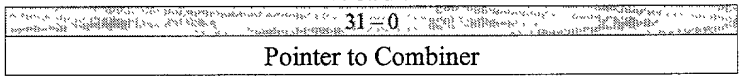
Given that the Combiner DSP software can be written by Morphics or the customer, the definition of the Combiner DSP messages cannot be known by the VMI. Effectively, this message is a 'wrapper' for the Combiner DSP messages.

Combiner DSP Msg Word 1 and *Combiner DSP Msg Word 2* are defined by whomever writes the Combiner DSP software. If Morphics defines these messages, a separate document will be provided to define all the supported messages that can be encoded within *Combiner DSP Msg Word 1* and *Combiner DSP Msg Word 2*.

Word 1 (Header Word)

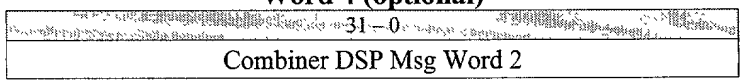


Word 2



Word 3

Word 4 (optional)



Combiner DSP Message Field	Description
Length	Length of this message, in 32-bit words, will be 3 or 4.
Msg Type	Always equal to COMBINER_DSP_MSG
Pointer to Combiner	Pointer to the Combiner associated with this message. It must be cast to (COMBINER *).
Combiner DSP Msg Word 1	First word of Combiner DSP message
Combiner DSP Msg Word 2	Second word of Combiner DSP message (optional, may not be sent)

11.2.1.2 Finger Offset Message Format

For CBME V1.05, the Combiner DSP does not have access to the finger offset. The VMI will use the Combiner DSP Message Queue to provide this information. In CBME V1.1, this information will be part of the Combiner DSP Message Format (see Section 11.2.1.1).

This message will be sent in response to the Finger_Request_Offset function (Section 10.1.7).

Word 1 (Header Word)

31 – 16	15 – 0
Msg Type (always FINGER_OFFSET_MSG)	Length (always 5)

Word 2

31 – 0
Pointer to Combiner

Word 3

31 – 0
Pointer to Finger

Word 4

31 – 0
Time Delay Offset

Word 5

31 – 0
Fractional Time Delay Offset

Finger Offset Message Field	Description
Length	Length of this message in 32-bit words. Always 5.
Msg Type	Always equal to FINGER_OFFSET_MSG
Pointer to Combiner	Pointer to the Combiner that the finger has been added to. It must be cast to (COMBINER *).
Pointer to Finger	Pointer to Finger that the offset is associated with. It must be cast to (FINGER *).
Time Delay Offset	See definition in *Finger Static Attributes Structure*, Section 10.1.3.1
Fractional Time Delay Offset	See definition in *Finger Static Attributes Structure*, Section 10.1.3.1

12 Downlink

The Downlink object represents the Primary Physical Tx channel.

12.1 Downlink Methods

CBME_New must always be called prior to taking any action on a Downlink object. This restriction will not be repeated for each method function.

12.1.1 Downlink_New

Prototype

UINT16 **Downlink_New**(CBME *p_cbme,
DOWNLINK *p_downlink,
CGU *p_cgu);

Description

Allocates a new Downlink object.

Input Parameters

p_cbme	pointer to parent CBME
p_downlink	pointer to Downlink object being allocated
p_cgu	pointer to CGU associated with this Downlink (see CGU, Section 4)

Restrictions

(Number of Downlinks allocated + Number of MTX's allocated) ≤ *max_downlinks* field in CBME resource attributes (see Section 3.1.12.1).

p_cgu must point to an initialized CGU that is a Downlink CGU (see Section 4.1.3.1). In addition, the CGU must be associated with the same CBME as the Uplink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.2 Downlink_Free

Prototype

UINT16 **Downlink_Free**(DOWNLINK *p_downlink);

Description

Deallocates a Downlink object.

Input Parameters

p_downlink pointer to the Downlink

Restrictions

Downlink_New must be called first.

There cannot be any MTX objects attached to the Downlink.

All Diversity antennas must be removed.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.3 Downlink_Set_Static_Attributes

Prototype

UINT16 **Downlink_Set_Static_Attributes(**
DOWNLINK *p_downlink,
DOWNLINK_STATIC_ATTRIB_TYPE *p_downlink_static_attrib);

Description

Sets the Downlink static attributes. See Section 12.1.3.1 for details on the static attributes. The Downlink maintains a copy of its attributes, so the static attribute structure passed in may be modified after this call.

Input Parameters

p_downlink	pointer to Downlink
p_downlink_static_attrib	pointer to Downlink static attributes

Restrictions

Downlink_New must be called first.

Downlink must not be running.

p_tpc_combiner field in static attributes must point to Combiner that belongs to same CBME as Downlink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.3.1 Downlink Static Attributes Structure

The 'C' structure is:

```
typedef struct  downlink_static_attrib_struct
{
   UINT16                        slot_format_index;
   UINT16                        antenna_num;
   UINT16                        channelization_number;
   UINT16                        frame_offset;
   COMBINER                      *p_tpc_combiner;
   M_DOWNLINK_FIELD_POWER_TYPE
field_power_levels[MAX_NUM_DOWNLINK_FIELDS];

}  DOWNLINK_STATIC_ATTRIB_TYPE;
```

Table 12-1 : Downlink Static Attributes

Downlink Static Attribute	Description
slot_format_index	Selects one of the slot formats downloaded via the scanchain. Valid Range: SLOT_FORMAT_MIN to SLOT_FORMAT_MAX The slot format index corresponds to the index in the CBME Downlink Slot Format List (see Section 3.1.15.1).
antenna_number	Antenna number of this channel. Valid Range: M_MIN_TX_ANTENNA to M_MAX_TX _ANTENNA
channelization_number	Must always be less than the spreading factor associated with the *slot_format_index*. For example, if the spreading factor is 128, the channelization number must be 0 – 127.
frame_offset	Frame offset in 256-chip intervals. Valid Range: M_DOWNLINK_MIN_FRAME_OFFSET to M_DOWNLINK_MIN_FRAME_OFFSET
p_tpc_combiner	Pointer to the Combiner (receive path) associated with this Downlink. This must be the Combiner associated with the channel containing TPC information.
field_power_levels	Array of field power levels. The indexes in this array correspond to the indexes of the multiplexed field types returned in *CBME_Get_Downlink_Field_List* (Section 3.1.14). The legal power ranges and fractional offsets are also contained in the Power List. See Section 12.1.3.1.1 for a description of this data type.

12.1.3.1.1 M_DOWNLINK_FIELD_POWER_TYPE

```
typedef struct downlink_field_struct
{
   UINT16   power_level;
   UINT16   fractional_offset;

} M_DOWNLINK_FIELD_POWER_TYPE;
```

Field	Description
power_level	Desired power level in whole dBs (e.g. 6). See *min_power* and *max_power* fields in Section 3.1.14.1. Valid Range: *min_power* to *max_power*
fractional_offset	Fractional power to add to the *power_level* field. See *fractional_range* field in Section 3.1.14.1. Valid Range: 0 to (*fractional_range* – 1)

12.1.4 Downlink_Start

Prototype UINT16 **Downlink_Start**(DOWNLINK *p_downlink)
Description Starts the Downlink channel.
Input Parameters **p_downlink** pointer to Downlink
Restrictions Downlink_Set_Static_Attributes must be called first. Downlink must not be running
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.5 Downlink_Stop

Prototype UINT16 **Downlink_Stop**(DOWNLINK *p_downlink)
Description Stop the Downlink channel.
Input Parameters **p_downlink** pointer to downlink
Restrictions Downlink must be running.
Return Values M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.6 Downlink_Add_Diversity

Prototype

UINT16 **Downlink_Add_Diversity(**

DOWNLINK	*p_downlink,
UINT16	antenna_num,
UINT8	diversity_type
UINT16	*p_tx_id);

Description

Adds diversity to a Downlink channel and to all MTX channels associated with the Downlink.

Input Parameters

p_downlink pointer to Downlink

antenna_num diversity channel antenna number
Valid Range: M_MIN_TX_ANTENNA to M_MAX_TX _ANTENNA

diversity_type M_STTD (currently supported)

p_tx_id pointer to where the ID for this diversity channel will be written (returned by function). This is the ID that will be time-multiplexed onto the CBME bus and will be used to interface the CODEC to the CBME.

If this function returns M_SUCCESS, the ID is valid.

Tx ID's for diversity channels can also be obtained from Downlink_Get_Diversity_List (see Section 12.1.13).

Restrictions

Downlink_Set_Static_Attributes must be called first.

The number of Diversity Antennas added to a Downlink must be ≤ M_MAX_DIVERSITY_PER_DOWNLINK

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.7 Downlink_Remove_Diversity

Prototype

UINT16 **Downlink_Remove_Diversity**(DOWNLINK *p_downlink,
UINT16 antenna_num)

Description

Removes diversity:

a) from the specified Downlink and

b) from all MTX channels associated with the Downlink.

Input Parameters

p_downlink	pointer to Downlink
antenna_num	diversity channel antenna number Valid Range: M_MIN_TX_ANTENNA to M_MAX_TX_ANTENNA

Restrictions

This diversity antenna must have been added to this Downlink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.8 Downlink_Add_MTX

Prototype

UINT16 **Downlink_Add_MTX**(DOWNLINK *p_downlink,
MTX *p_mtx);

Description

Adds an MTX (multi-code channel) to a Downlink.

Input Parameters

p_downlink pointer to Downlink
p_mtx pointer to MTX being added to Downlink

Restrictions

MTX_New must have been called first.

MTX must not already be added to a Downlink.

The number of MTX's added to a Downlink must be ≤ M_MAX_MTX_PER_DOWNLINK.

The MTX and Downlink must belong to the same CBME.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.9 Downlink_Remove_MTX

Prototype

UINT16 **Downlink_Remove_MTX**(MTX *p_mtx);

Description

Removes an MTX from a Downlink along with any diversity channels associated with this MTX.

Input Parameters

p_mtx pointer to MTX being removed

Restrictions

Downlink_Add_MTX must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.10 Downlink_Get_Tx_ID

Prototype

UINT16 **Downlink_Get_Tx_ID**(DOWNLINK *p_downlink,
UINT16 *p_tx_id);

Description

Retrieves the transmitter ID for the primary channel associated with the Downlink. This is the ID that will be time-multiplexed onto the CBME bus and will be used to interface the CODEC to the CBME.

Input Parameters

p_downlink pointer to downlink
p_tx_id pointer to where Tx ID for Downlink will be written.

Restrictions

Downlink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.11 Downlink_Get_Static_Attributes

Prototype

UINT16 **Downlink_Get_Static_Attributes**(
DOWNLINK *p_downlink,
DOWNLINK _STATIC_ATTRIB_TYPE *p_static_att_table);

Description

Retrieves the transmitter static attributes and copies them to the user-supplied structure.

Note that this function returns the values that were set when Downlink_Set_Static_Attributes() was called. Use the Downlink_Request_Field_Power_Levels function to retrieve the current state of the attributes. For example, a downlink channel may dynamically update the field power levels; in order to read the current state of the field power levels, call Downlink_Request_Field_Power_Levels().

Input Parameters

p_downlink pointer to downlink
p_static_att_table pointer to table where attributes will be copied.

Restrictions

Downlink_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.12 Downlink_Get_MTX_List

Prototype

UINT16 **Downlink_Get_MTX_List**(DOWNLINK *p_downlink,
M_MTX_LIST_TYPE *p_mtx_list);

Description

Fills in the list with MTXs that have been added to this Downlink. The list is filled in starting from index 0 to (num MTXs – 1). Unused entries are nulled out.

Input Parameters

p_downlink pointer to Downlink

p_mtx_list pointer to MTX list where MTX pointers are written (see Section 12.1.12.1 for the definition of the M_MTX_LIST_TYPE).

Restrictions

Downlink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.12.1 M_MTX_LIST_TYPE

```
typedef struct mtx_list_struct
{
  UINT16      num_mtx;
  MTX        *p_mtx[M_MAX_MTX_PER_DOWNLINK];

} M_MTX_LIST_TYPE;
```

Field	Description
num_mtx	Number of MTXs in the list.
p_mtx	List of pointers to MTX objects added to this downlink. If *num_mtx* > 0, then the valid range is 0 to (*num_mtx* – 1).

12.1.13 Downlink_Get_Diversity_List

Prototype

UINT16 **Downlink_Get_Diversity_List**(
DOWNLINK *p_downlink,
M_DIVERSITY_LIST_TYPE *p_div_list);

Description

Fills in the list with antenna numbers and corresponding diversity type that have been added to this Downlink. Unused entries in the list have the *used* flag set to M_FALSE.

Input Parameters

p_downlink	pointer to Downlink
p_div_list	pointer to Diversity list where information is written. See Section 12.1.13.1.

Restrictions

Downlink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.13.1 Downlink_Diversity_List

Two structures, one defining a list element, and one defining the list itself comprise the Downlink Diversity List.

```
typedef struct diversity_struct
{
   UINT16  used;

   UINT16  antenna_number;

   UINT16  diversity_type;

   UINT16  tx_id;

} M_DIVERSITY_TYPE;
```

Field	Description
used	M_TRUE if this list location is has valid data, else M_FALSE.
ant_number	Diversity antenna number for this diversity channel
diversity_type	Diversity type for this diversity channel. Valid Range: currently, only M_STTD
tx_id	Transmitter ID for this Diversity channel

```
typedef struct diversity_list_struct
{
   UINT16  num_diversity;

   M_DIVERSITY_TYPE  diversity[M_MAX_DIVERSITY_PER_DOWNLINK];

} M_DIVERSITY_LIST_TYPE;
```

Field	Description
num_diversity	Number of valid Diversity elements in the list Valid Range: 0 to M_MAX_DIVERSITY_PER_DOWNLINK
diversity	Array of diversity elements in the list

12.1.14 Downlink_Get_State

Prototype

UINT16 **Downlink_Get_State**(DOWNLINK *p_downlink,
UINT16 *p_downlink_state);

Description

Gets a Downlink state (running or stopped)

Input Parameters

p_downlink	pointer to the Downlink
p_downlink_state	pointer to where Downlink state is written (M_DOWNLINK_RUNNING or M_DOWNLINK_STOPPED)

Restrictions

Downlink_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

12.1.15 Downlink_Get_Associated_CGU

Prototype

CGU * **Downlink_Get_Associated_CGU**(DOWNLINK *p_downlink,
UINT16 *p_error_code)

Description

Returns pointer to the CGU associated with this Downlink.

Input Parameters

p_downlink	pointer to the Downlink
p_error_code	pointer where error code is written

Restrictions

Downlink_New must be called first.

Return Values

(a) valid pointer to associated CGU and *p_error_code = M_SUCCESS

or

(b) NULL and *p_error_code contains an error code (see Section 14.1)

12.1.16 Downlink_Set_User_Data

Prototype

```
UINT16 Downlink_Set_User_Data( DOWNLINK  *p_downlink,
                               UINT16    index,
                               UINT16    length,
                               UINT8     *p_data);
```

Description

See Section 14.3.1 for description of this function.

12.1.17 Downlink_Get_User_Data

Prototype

```
UINT16 Downlink_Get_User_Data( DOWNLINK  *p_downlink,
                               UINT16    index,
                               UINT16    length,
                               UINT8     *p_data);
```

Description

See Section 14.3.2 for description of this function.

13 MTX (Multicode Tx Channel)

Multicode channels can optionally be added to a Downlink. The Multicode Tx (MTX) object is always associated with a Downlink object. One or more MTXs can be associated with a Downlink.

13.1 MTX Methods

Where a Downlink object is referenced, Downlink_New must always be called prior to taking any action on a MTX. This restriction will not be repeated for each method function.

13.1.1 MTX_New

Prototype

UINT16 **MTX_New**(CBME *p_cbme,
MTX *p_mtx);

Description

Allocates a new MTX object.

Input Parameters

p_cbme pointer to parent CBME
p_mtx pointer to MTX object being allocated

Restrictions

(Number of Downlinks allocated + Number of MTX's allocated) ≤ *max_downlinks* field in CBME resource attributes (see Section 3.1.12.1).

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

13.1.2 MTX_Free

Prototype

UINT16 **MTX_Free**(MTX *p_mtx);

Description

Deallocates a MTX object.

Input Parameters

p_mtx pointer to the MTX

Restrictions

MTX_New must be called first.

The MTX cannot be attached to a Downlink.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

13.1.3 MTX_Set_Static_Attributes

Prototype

```
UINT16 MTX_Set_Static_Attributes(
            MTX                         *p_mtx,
            MTX_STATIC_ATTRIB_TYPE      *p_mtx_static_attrib);
```

Description

Sets the MTX static attributes. See Section 13.1.3.1 for details on the static attributes. The MTX maintains a copy of its attributes, so the static attribute structure passed in may be modified after this call.

Input Parameters

p_mtx pointer to MTX

p_mtx_static_attrib pointer to MTX static attributes

Restrictions

MTX_New must be called first.

If MTX has been added to a Downlink, the Downlink must not be running.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

13.1.3.1 MTX Static Attributes Structure

The 'C' structure is:

```
typedef struct  mtx_static_attrib_struct
{
        UINT16                                  channelization_number;
        M_DOWNLINK_FIELD_POWER_TYPE
field_power_levels[MAX_NUM_DOWNLINK_FIELDS];

} MTX_STATIC_ATTRIB_TYPE;
```

Table 13-1 : Multicode Static Attributes

MTX Static Attribute	Description
channelization_number	Must always be less than the spreading factor associated with the *slot_format_index* of the parent Downlink. (sese Section 12.1.3.1). For example, if the spreading factor is 128, the channelization number must be 0 – 127.
field_power_levels	Array of field power levels. The indexes in this array correspond to the indexes of the multiplexed field types returned in *CBME_Get_Downlink_Field_List* (Section 3.1.14). The legal power ranges and fractional offsets are also contained in the Power List. See Section 12.1.3.1.1 for a description of this data type.

13.1.4 MTX_Get_Static_Attributes

Prototype

UINT16 **MTX_Get_Static_Attributes(**
MTX *p_mtx,
MTX_STATIC_ATTRIB_TYPE *p_static_att_table);

Description

Retrieves the MTX static attributes and copies them to the user-supplied table.

Input Parameters

p_mtx	pointer to MTX
p_static_att_table	pointer to table where attributes will be copied.

Restrictions

MTX_Set_Static_Attributes must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

13.1.5 MTX_Get_Tx_ID

Prototype

UINT16 **MTX_Get_Tx_ID**(MTX *p_mtx,
UINT16 *p_tx_id);

Description

Retrieves the transmitter ID for this Multicode channel. This is the ID that will be time-multiplexed onto the CBME bus and will be used to interface the CODEC to the CBME.

Input Parameters

p_mtx	pointer to MTX
p_tx_id	pointer to where Tx ID for MTX will be written.

Restrictions

MTX_New must be called first.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

13.1.6 MTX_Set_User_Data

Prototype

```
UINT16 MTX_Set_User_Data( MTX      *p_mtx,
                          UINT16   index,
                          UINT16   length,
                          UINT8    *p_data);
```

Description

See Section 14.3.1 for description of this function.

13.1.7 MTX_Get_User_Data

Prototype

```
UINT16 MTX_Get_User_Data(MTX      *p_mtx,
                         UINT16   index,
                         UINT16   length,
                         UINT8    *p_data);
```

Description

See Section 14.3.2 for a description of this function.

14 Appendix

14.1 VMI Error Codes

Table 14-1 : VMI Error Codes

Error Code Define	Numeric Value	Description
General Purpose Error Codes		
M_SUCCESS	0x0000	Operation was successful.
M_INVALID_ANT_PORT_ERROR	0x0001	Invalid antenna port.
M_USER_INDEX_ERROR	0x0002	Trying to access an area beyond the allocated user area. Index or length, or combination of them, is invalid.
M_ACTION_NOT_SUPPORTED	0x0003	This action is not supported by this version of the VMI.
M_RTOS_MSG_QUEUE_CREATE_ERROR	0x0004	VMI call to VMI_Msg_Queue_Create returned an error (see Section 2.3.3)
CBME Error Codes		
M_CBME_COMM_ERRROR	0x0101	Unable to communicate with CBME
M_CBME_NEW_CBME_ERROR	0x0102	*CBME_New* must be called before performing this action.
M_CBME_INVALID_MERGE_ERROR	0x0103	merge_interrupt_action flag is invalid.
M_CBME_FINGER_BLOCK_SIZE_ERROR	0x010B	Finger block size must be 4, 6, or 8.
M_CBME_NUM_MOBILES_ERROR	0x010C	num_mobiles * finger_block_size is greater than the total number of tracking fingers available.
M_CBME_NO_MOBILE_RESOURCES_ERRO R	0x010D	Must call CBME_Set_Mobile_Resources() before calling this function.
M_CBME_NO_SEARCHER_TIME_PERIOD	0x010E	Must call CBME_Set_Search_Periodicity() before calling this function.
M_CBME_SUBCHIP_PHASE_ERROR	0x010F	One or more of the subchip phase counts are out of range.
M_CBME_DSM_SUBCHIP_NOT_INIT_ERROR	0x0110	Must call CBME_Set_DSM_Subchip_Phase() before calling this function.
M_CBME_NUM_ACCESS_SLOTS_ERROR	0x0111	Number of access slots must be ≤ M_MAX_PDE
M_CBME_NUM_TIME_SLOTS_ERROR	0x0112	(num_access_slots * num_access_slot_sets) must be ≤ M_MAX_PDE
M_CBME_NO_RAM_SCANCHAIN_ERROR	0x0113	Cannot perform this operation until the RAM scanchain has been downloaded.

Error Code Define	Numeric Value	Description
M_CBME_NO_REG_SCANCHAIN_ERROR	0x0114	Cannot perform this operation until register scanchains has been downloaded.
M_CBME_SEARCHER_SCALING_ERROR	0x0115	*scale_value* field out of range.
M_CBME_NO_SEARCHER_SCALE_ERROR	0x0116	Cannot perform this operation until CBME_Set_Searcher_Energy_Scaling() is called.
Searcher Error Codes		
M_ SEARCHER_LIMIT_ERROR	0x0200	Maximum number of searchers already allocated. Cannot allocate any more.
M_SEARCHER_NEW_SEARCHER_ERROR	0x0202	*Searcher_New* must be called before performing this action.
M_SEARCHER_NO_DSM_ERROR	0x0203	No Searcher DSM has been assigned to this searcher. Cannot perform this action until Searcher DSM assigned to Searcher.
M_ SEARCHER_INVALID_TYPE_ERROR	0x0206	This operation is being performed on the wrong kind of searcher ('new mobile' or 'existing mobile').
M_SEARCHER_WINDOW_SIZE_ERROR	0x0207	Invalid searcher window size
M_SEARCHER_RUNNING_ERROR	0x0208	Cannot perform this action while the searcher is running.
M_SEARCHER_NO_STATICS_ERROR	0x0209	This operation cannot be performed because the searcher does not have valid static attributes.
M_SEARCHER_ALREADY_RUNNING_ERRO R	0x020B	Can't start a searcher that is already running.
M_SEARCHER_ALREADY_STOPPED_ERROR	0x020C	Can't stop a searcher that is already stopped.
M_SEARCHER_DSM_MISMATCH_ERROR	0x020D	Searcher and DSM belong to different CBMEs.
M_SEARCHER_STOPPED_ERROR	0x020E	Cannot perform this action while the searcher is stopped.
M_SEARCHER_SAME_SEARCHER_ERROR	0x020F	Cannot copy a searcher to itself.
M_SEARCHER_NO_UPLINK_ERROR	0x0210	Cannot perform this action until searcher has been added to an Uplink object.
M_SEARCHER_NO_FINGER_ERROR	0x0212	Cannot perform this action until the uplink has had one finger added to it. See restrictions in *Searcher_Start.*
M_SEARCHER_PILOT_GATE_ERROR	0x0213	Invalid value for pilot_gating parameter.
M_SEARCHER_INVALID_OFFSET_ERROR	0x0215	Invalid *start_search_offset* parameter.
M_SEARCHER_INVALID_DSM_ERROR	0x0216	Cannot perform this operation with a Searcher DSM that has invalid state attributes.
M_SEARCHER_CGU_MISMATCH_ERROR	0x0217	'new mobile' searcher and CGU don not belong to the same CBME.
M_SEARCHER_EXCEED_ANT_BUF_ERROR	0x0218	Search window size and searcher offset are greater than the antenna buffer (see *start_search_offset* field in Section 6.1.3.1)

Error Code Define	Numeric Value	Description
Searcher DSM Error Codes		
M_DSM_LIMIT_ERROR	0x0300	Maximum number of DSMs allocated. Cannot allocate any more.
M_DSM_NEW_DSM_ERROR	0x0302	*New_DSM* must be called before performing this action.
M_DSM_ INT_LENGTH_ERROR	0x0303	Invalid DSM integration length.
M_DSM_ THRESHOLD_ERROR	0x0305	Invalid DSM threshold.
M_DSM_PDI_LENGTH_ERROR	0x0306	Invalid PDI length.
M_DSM_GDS_ERROR	0x0307	Invalid gds.
M_DSM_NOT_COMPLETE_ERROR	0x0309	DSM state machine allocated with *n* states and not all of them are defined.
M_DSM_STATIC_ATTRIB_ERROR	0x030A	This DSM static attributes have not been defined.
M_DSM_NO_SUBCHIP_PHASE_ERROR	0x030E	Cannot allocate a Searcher_DSM until *CBME_Set_Subchip_Phase* has been called.
Finger Error Codes		
M_FINGER_LIMIT_ERROR	0x0400	Maximum number of fingers has been allocated. Unable to allocate another finger.
M_FINGER_NEW_FINGER_ERROR	0x0401	*Finger_New* must be called before performing this action.
M_FINGER_RUNNING_ERROR	0x0402	Cannot perform this action while the finger is running.
M_FINGER_TIME_DELAY_OFFSET_ERROR	0x0403	*time_delay_offset* must be ≤ *uplink_antenna_buffer_size*. See Section 10.1.3.1)
M_FINGER_ATTACHED_COMBINER_ERROR	0x0404	Cannot perform this action when the finger has been added to a combiner.
M_FINGER_NO_STATICS_ERROR	0x0408	*Finger_Set_Static_Attributes* must be called before performing this action.
M_FINGER_STOPPED_ERROR	0x040B	Cannot perform this action while the finger is stopped.
M_FINGER_SAME_FINGER_ERROR	0x040C	Cannot copy the same finger to itself.
M_FINGER_NO_COMBINER_ERROR	0x040E	Cannot perform this action on a finger that has not been added to a combiner.
M_FINGER_COMBINER_STOPPED_ERROR	0x040F	Cannot start a finger when its associated combiner is stopped.
M_FINGER_TIME_DELAY_OFFSET_ERROR	0x0415	Invalid time delay offset. See Section 10.1.3.1.
M_FINGER_FRACT_TIME_DELAY_OFFSET_ ERROR	0x0416	Invalid fractional time delay offset. . See Section 10.1.3.1.
Preamble Detection Engine Error Codes		
M_PDE_LIMIT_ERROR	0x0500	Maximum number of preamble detection engines has been allocated; cannot allocate another preamble detection engine.

Error Code Define	Numeric Value	Description
M_PDE_NEW_PDE_ERROR	0x0501	*PDE_New* must be called before performing this action.
M_PDE_RUNNING_ERROR	0x0502	This action cannot be performed when the preamble detection engines are running
M_PDE_ACCESS_SLOT_ERROR	0x0503	*CBME_Set_PDE_Num_Slots* must be called first.
M_PDE_NONE_DEFINED_ERROR	0x0504	At least one Preamble Detection Engine must be defined before this operation.
M_PDE_TIMESLOT_ERROR	0x0505	*time_slot_num* invalid; too large. See Section 8.
M_PDE_MODE_ERROR	0x0506	Invalid Preamble Detection Engine *mode.*
M_PDE_FORCE_ENABLE_ERROR	0x0507	Invalid value for *force_results_flag.*
M_PDE_NUM_REPORTS_ERROR	0x0508	Invalid value for *num_reports.*
M_PDE_ENERGY_SCALE_ERROR	0x0509	Invalid value for *energy_scale*
M_PDE_THESHOLD_SCALE_ERROR	0x050A	Invalid value for *threshold_scale.*
M_PDE_STOPPED_ERROR	0x050B	This action cannot be performed while the preamble detection engines are stopped.
M_PDE_TIMESLOT_IN_USE_ERROR	0x050C	The time slot is already assigned to another Preamble Detection Engine.
M_PDE_INVALID_STATIC_ERROR	0x050E	This action cannot be performed until *PDE_Set_Static_Attributes* is called.
M_PDE_ATTACHED_ANT_ERROR	0x050F	Cannot perform this action on a PDE with antennas attached to it.
M_PDE_ANT_LIMIT_ERROR	0x0510	Preamble Detection Engine already has maximum antennas for its mode.
M_PDE_ENERGY_UNDERFLOW_ERROR	0x0511	Attempted to read a preamble detection engine result from an empty queue.
M_PDE_ENERGY_OVERFLOW_ERROR	0x0512	Preamble Detection Engine queue is full, and energy results are being thrown away.
M_PDE_PDE_ANT_MISMATCH_ERROR	0x0513	Preamble Detection Engine and Preamble Detection Engine Antenna do not belong to the same CBME.
M_PDE_ANTENNA_ERROR	0x0514	This Preamble Detection Engine Antenna is not currently added to this Preamble Detection Engine.
Preamble Detection Engine Antenna Error Codes		
M_PDE_ANT_NEW_PDE_ANT_ERROR	0x0581	*PDE_Ant_New* must be called before performing this action.
M_PDE_ANT_INVALID_ANT_NUM_ERROR	0x0582	Invalid value for *ant_num.*
M_PDE_ANT_INVALID_PHASE_ERROR	0x0583	Invalid value for *ant_phase_select.*
M_PDE_ANT_IN_USE_ERROR	0x0584	The antenna number is already assigned to another Preamble Detection Engine Antenna.
M_PDE_ANT_INVALID_STATIC_ERROR	0x0585	This action cannot be performed until *PDE_Ant_Set_Static_Attributes* is called.

Error Code Define	Numeric Value	Description
M_PDE_ANT_ATTACHED_PDE_ERROR	0x0586	Cannot perform this action on a PDE Antenna with a Preamble Detection Engine attached to it.
M_PDE_ANT_CGU_MISMATCH_ERROR	0x0587	PDE Antenna and CGU must belong to the same CBME.
Combiner Error Codes		
M_COMBINER_LIMIT_ERROR	0x0600	Maximum number of combiners already allocated; cannot allocate another combiner.
M_COMBINER_FINGER_LIMIT_ERROR	0x0601	Maximum number of fingers already added to this combiner. No more fingers can be added.
M_COMBINER_NEW_COMBINER_ERROR	0x0602	*Combiner_New* must be called before performing this action.
M_COMBINER_RUNNING_ERROR	0x0603	This action cannot be performed while the combiner is running.
M_ COMBINER_FINGER_MISMATCH_ERROR	0x0604	Finger and combiner must belong to same CBME.
M_COMBINER_FRAME_OFFSET_ERROR	0x0605	Invalid *frame_offset*.
M_COMBINER_FINGER_ADD_ERROR	0x0606	Finger being added has already has been added to another combiner.
M_COMBINER_NO_FINGER_ERROR	0x0607	Cannot perform this action on a combiner with no fingers added to it.
M_COMBINER_ALREADY_RUNNING_ERRO R	0x0608	Cannot start a combiner that is already running.
M_COMBINER_ALREADY_STOPPED_ERRO R	0x0609	Cannot stop a combiner that is already stopped.
M_COMBINER_FREE_ERROR	0x060B	Cannot free a combiner that still has fingers associated with it.
M_COMBINER_FINGER_ADD_ERROR	0x060C	Cannot perform this action; this finger has already been added to a combiner.
M_COMBINER_NO_PERM_BLOCKS_ERROR	0x060D	No more permanent finger blocks available. See Section 14.2.
M_COMBINER_NO_UPLINK_ERROR	0x0610	Cannot perform this action until combiner has been added to an Uplink object.
M_COMBINER_UPLINK_ERROR	0x0611	Cannot free a combiner that is still attached to an uplink.
M_COMBINER_SPREAD_FACTOR_ERROR	0x0612	Invalid *spreading_factor* (see Section 11.1.3.1).
M_COMBINER_CHANNEL_NUMBER_ERROR	0x0613	Invalid *channelization_number* (see Section 11.1.3.1).
M_COMBINER_NO_STATICS_ERROR	0x0614	*Combiner_Set_Static_Attributes* must be called before performing this action.
Downlink Error Codes		
M_DOWNLINK_LIMIT_ERROR	0x0B00	Maximum number of Downlinks already allocated; cannot allocate another Downlink
M_DOWNLINK_NEW_DOWNLINK_ERROR	0x0B01	Downlink_New must be called before first.

Error Code Define	Numeric Value	Description
M_DOWNLINK_RUNNING_ERROR	0x0B02	Cannot perform this action when Downlink is running.
M_DOWNLINK_COMBINER_MISMATCH_ERROR	0x0B03	Downlink and Combiner must belong to same CBME.
M_DOWNLINK_ANTENNA_ERROR	0x0B04	*antenna_number* field is out of range.
M_DOWNLINK_SLOT_FORMAT_ERROR	0x0B05	*slot_format_index* field out of range.
M_DOWNLINK_CHAN_NUMBER_ERROR	0x0B07	*channelization_number* must be smaller than the spreading factor. See Table 12-1.
M_DOWNLINK_COMBINER_ERROR	0x0B08	Associated Combiner must have at least one finger and be running.
M_DOWNLINK_POWER_ERROR	0x0B09	One or more power levels or fractional power levels are out of range. See Section 3.1.14.1.
M_DOWNLINK_NO_STATICS_ERROR	0x0B0A	*Downlink_Set_Static_Attributes* must be called before performing this action.
M_DOWNLINK_FRAME_OFFSET_ERROR	0x0B0B	*frame_offset* field out of range.
M_DOWNLINK_ALREADY_RUNNING_ERROR	0x0B0D	Downlink is already running.
M_DOWNLINK_ALREADY_STOPPED_ERROR	0x0B0E	Downlink is already stopped.
M_DOWNLINK_MTX_LIMIT_ERROR	0x0B0F	Cannot add any more MTXs to this Downlink; currently at limit.
M_DOWNLINK_DIVERSITY_ERROR	0x0B10	*diversity* field is invalid.
M_DOWNLINK_MAX_DIVERSITY_ERROR	0x0B11	Cannot add any more diversity antennas to this Downlink; currently at maximum.
M_DOWNLINK_INVALID_DIVERSITY_ERROR	0x0B12	Diversity antenna is not in Downlink list of diversity antennas.
M_DOWNLINK_MTX_ATTACHED_ERROR	0x0B14	Cannot perform this action if any MTX objects are still attached to Downlink.
M_DOWNLINK_DIVERSITY_ATTACHED_ERROR	0x0B15	Cannot perform this action if any Diversity Antennas are still added to the Downlink.
M_DOWNLINK_CGU_MISMATCH_ERROR	0x0B16	Downlink and CGU must belong to the same CBME.
M_DOWNLINK_MTX_MISMATCH_ERROR	0xB17	Downlink and MTX must belong to the same CBME.
MTX Error Codes		
M_MTX_NEW_MTX_ERROR	0x0C01	MTX_New must be called before first.
M_MTX_ALREADY_ADDED_ERROR	0x0C02	MTX already added to a downlink.
M_MTX_NOT_IN_LIST_ERROR	0x0C03	MTX has not been added to a downlink.
M_MTX_INVALID_STATICS	0x0C04	*MTX_Set_Static_Attributes* must be called before performing this action.
M_MTX_DOWNLINK_ATTACHED_ERROR	0x0C05	Cannot perform this action if MTX is still added to a downlink.
Uplink Error Codes		

Error Code Define	Numeric Value	Description
M_UPLINK_NEW_UPLINK_ERROR	0x0801	*Uplink_New* must be called before performing this action.
M_UPLINK_COMBINER_LIMIT_ERROR	0x0802	Maximum number of combiners already added to this uplink; cannot add another combiner.
M_UPLINK_SEARCHER_LIMIT_ERROR	0x0803	Maximum number of searchers already added to this uplink; cannot add another searcher.
M_UPLINK_EXIST_MOB_SEARCH_ERROR	0x0804	Can only add an 'existing mobile' searcher to an uplink. See 6.1.1, page 69.
M_UPLINK_SLOT_FORMAT_INIT_ERROR	0x0805	Cannot perform this action until Uplink_Set_DPCCH_Slot_Format has been called (see Section 5.1.3)
M_UPLINK_OBJECTS_ATTACHED_ERROR	0x0806	This operation cannot be performed while objects are still attached to the uplink.
M_UPLINK_NO_COMBINER_ERROR	0x0807	This operation cannot be performed until at least one combiner has been added to the uplink.
M_UPLINK_SLOT_FORMAT_ERROR	0x0808	Invalid slot number.
M_UPLINK_CGU_MISMATCH_ERROR	0x0809	Uplink and CGU do not belong to the same CBME.
M_UPLINK_COMBINER_MISMATCH_ERROR	0x080A	Uplink and Combiner do not belong to the same CBME.
M_UPLINK_SEARCHER_MISMATCH_ERROR	0x080B	Uplink and Searcher do not belong to the same CBME.
Scanchain Error Codes		
M_SCANCHAIN_INVALID_TYPE_ERROR	0x0900	Invalid Scanchain type.
CGU Error Codes		
M_CGU_NEW_CGU_ERROR	0x0A00	CGU_New must be called before first.
M_CGU_ZERO_INSERTION_ERROR	0x0A01	*zero_insertion_enable* must be M_TRUE or M_FALSE.
M_CGU_IN_USE_ERROR	0x0A02	At least one object (searcher, Preamble Detection Engine, Combiner, etc.) is still using this CGU.
M_CGU_INVALID_CGU_INDEX_ERROR	0x0A03	The CGU index is out of range with respect the number of CGUs that were downloaded via the scanchain. See *num_on_chip_cgus* field in Section 3.1.13.1.
M_CGU_OBJECT_MISMATCH_ERROR	0x0A04	The CGU object type does not match the object being created.

14.2 Number of Mobiles and Finger Blocks

This section describes the implications of what is going on 'under the hood' in terms of how CBME_Set_Mobile_Resources() function (see Section 3.1.5) affects resources and performance.

Presume the following sequence of calls takes place:

CBME_New()
CBME_Get_Resource_Attributes()

After calling CBME_Get_Resource_Attributes(), presume the *max_fingers* attribute is 1536. This means that the CBME can support a maximum of 1536 tracking fingers.

CBME_Set_Mobile_Resources() has two parameters that are of interest, *finger_block_size* and *num_mobiles*.

The *finger_block_size* indicates how many tracking Fingers are initially allocated to a Combiner when it is created via Combiner_New(). This block of fingers is also known as the combiners 'permanent' block. The combiner, as long as its allocated, keeps this block of fingers, whether they are being used or not. The size of the permanent block affects performance. If for example, the *finger_block_size* is 4, then adding the first 4 fingers, and removing any of these fingers, is a relatively fast operation. If a 5th finger needs to be added to the combiner, a finger will have to taken from the 'reserve' pool, which is a slower operation.

The *num_mobiles* indicates the maximum number of Combiner objects that can be allocated for a CBME. Since each mobile being tracked requires at least one Combiner, this value reflects the maximum number of mobiles the CBME will be able to support. Note that the *max_combiners* field in the CBME resource attributes indicates the total number of combiners the CBME is capable of supporting. Therefore, $num\_mobiles \leq max\_combiners$.

From a Combiners perspective, there are two types of fingers: those fingers in its 'permanent' block (dictated by the *finger_block_size* parameter), and fingers that are allocated from the reserve pool. As long as the Combiner is allocated, the fingers in the permanent block are 'tied' to that combiner, whether they are used or not. Fingers from the reserve pool always come in pairs. So, if the *finger_block_size* is 4, and a combiner needed a 5th finger, it would automatically get two fingers from the reserve pool. One of these would immediately be used, the other is available in case a 6th finger is needed. When both 'reserve pool' fingers are no longer being used, they are returned to the reserve pool.

Again, the usage of fingers from the permanent block, and allocating fingers from the reserve block, are operations performed by the VMI library. The user simply calls Combiner_Add_Finger() and Combiner_Remove_Finger(). The VMI library performs these operations in an optimized manner.

The tradeoff that needs to be considered is system performance (speed of adding and removing fingers) versus system flexibility (number of mobiles supported). It is faster to use fingers in the combiners 'permanent' block of fingers. It is slower to get fingers from the reserve pool.

Table 14-2 shows several different scenarios for configuring the number of mobiles and the finger block size. Each of these is discussed in terms of performance versus flexibility.

Table 14-2 : Usage of *num_mobiles* and *finger_block_size*

Example #	*num_mobiles*	*finger_block_Size*	Fingers pre-allocated	Reserve Fingers
1	384	4	1536	0 (0 pairs)
2	300	4	1200	336 (168 pairs)
3	100	8	800	736 (368 pairs)
4	250	6	1500	36 (18 pairs)

In Example 1, the CBME's 1536 fingers are used to support a total of 384 mobiles. There are no reserve fingers left over; thus, no combiner can have more than 4 fingers. This configuration yields the maximum number of mobiles supported, but at the expense of no reserve fingers. Finger adding/removing is always within the permanent block, which is relatively fast.

In Example 2, the CBME supports 300 mobiles, each with a minimum of 4 fingers. This used up 1200 of the 1536 fingers, leaving 336 fingers in reserve. The reserve fingers can be used for combiners that need more than 4 fingers. So, compared to Example 1, this setup is more flexible in that combiners that need more than 4 fingers can allocate them from the reserve pool.

In Example 3, the CBME supports 100 mobiles, each with a minimum of 8 fingers. This uses 800 of the 1536 fingers, leaving 736 fingers in reserve. Here, the CBME is supporting far fewer than the maximum number of mobiles, but with 8 fingers per combiner, adding and removing fingers is very efficient.

In Example 4, the CBME supports 250 mobiles, each with a minimum of 6 fingers. This leaves 36 fingers in the reserve pool. Since fingers from the reserve pool come in pairs, there are effectively 18 pairs that can be allocated from the reserve pool. With 6 fingers per mobile, adding and removing fingers is fast, but there are not many reserve fingers available.

14.3 object_Set_User_Data and object_Get_User_Data

The set and get functions behave the same for all objects. They allow the user to store and retrieve application data from any CBME object (CBME, Searcher, Searcher DSM, Preamble Detection Engine, Finger, Combiner, Uplink, etc.). The number of user bytes allocated for each type of CBME object is determined at compile time by the following defines in **cbme.h**:

```
#define NUM_CBME_USER_BYTES                1
#define NUM_SEARCHER_USER_BYTES            1
#define NUM_SEARCHER_DSM_USER_BYTES        1
#define NUM_PDE_USER_BYTES                 1
#define NUM_PDE_ANT_USER_BYTES             1
#define NUM_FINGER_USER_BYTES              1
#define NUM_COMBINER_USER_BYTES            1
#define NUM_TX_USER_BYTES                  1
#define NUM_UPLINK_USER_BYTES              1
#define NUM_CGU_USER_BYTES                 1
#define NUM_DOWNLINK_USER_BYTES            1
#define NUM_MTX_USER_BYTES                 1
```

The default (and minimum size) is 1 byte of user data per object type. The size of the user data can theoretically be set to any desired value. The only constraints are the amount of available system memory. User data is zeroed out when the object is freed (e.g. Finger_Free, Searcher_Free, etc.).

The following two sections describe the two functions (for each object) used to read and write object user data.

14.3.1 *object*_Set_User_Data

Prototype

UINT16 ***object*_Set_User_Data**(*object* *p_object,
UINT16 index,
UINT16 length,
UINT8 *p_data);

Description

Writes user data to an object.

Input Parameters

p_object	pointer to object (e.g. FINGER, COMBINER, etc.)
index	index into object user data to start writing to (0 to (size – 1))
length	number of bytes to write to object
p_data	pointer to source data

Restrictions

*object*_New must be called first

index + *length* must not exceed the size of the user data as defined in cbme.h.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

14.3.2 *object*_Get_User_Data

Prototype

UINT16 ***object*_Get_User_Data**(*object* *p_object,
UINT16 index,
UINT16 length,
UINT8 *p_data);

Description

Reads user data from an object.

Input Parameters

p_object	pointer to object (e.g. FINGER, COMBINER, etc.)
index	index in object user data to start reading from (0 to (size – 1))
length	number of bytes to read from object
p_data	pointer to where the user data will be written

Restrictions

*object*_New must be called first

index + *length* must not exceed the size of the user data as defined in cbme.h.

Return Values

M_SUCCESS or error code (see Section 14.1 for error codes)

14.4 Preamble Detection Engine Modes

The Preamble Detection Engine can be time-shared across 24 antennas in many different modes. Table 14-3 shows the modes that are supported for a 3.84 MHz Chip Rate:

Table 14-3 : Preamble Detection Engine Supported Modes for 3.84 MHz Chip Rate

Mode	Rate	# Ant	# Taps	# Hypothesis	OQPSK♦
1 Antenna, 1x					
0	1x	1	2048	1024	no
1	1x	1	1920	1152	no
2	1x	1	1792	1280	no
3	1x	1	1664	1408	no
4	1x	1	1536	1536	no
5	1x	1	1408	1664	no
6	1x	1	1280	1792	no
7	1x	1	1152	1920	no
8	1x	1	1024	2048	no
9	1x	1	896	2176	no
10	1x	1	768	2304	no
11	1x	1	640	2432	no
12	1x	1	512	2560	no
13	1x	1	384	2688	no
14	1x	1	256	2816	no
15	1x	1	128	2944	no
1 Antenna, 2x					
16	2x	1	1408	128	supported
17	2x	1	1280	256	supported
18	2x	1	1152	384	supported
19	2x	1	1024	512	supported
20	2x	1	896	640	supported
21	2x	1	768	768	supported
22	2x	1	640	896	supported
23	2x	1	512	1024	supported
24	2x	1	384	1152	supported
25	2x	1	256	1280	supported
26	2x	1	128	1408	supported
2 Antennas, 1x					
27	1x	2	1408	128	no
28	1x	2	1280	256	no
29	1x	2	1152	384	no
30	1x	2	1024	512	no

♦ OQPSK must always be off for 3GPP

Mode	Rate	# Ant	# Taps	# Hypothesis	OQPSK*
31	1x	2	896	640	no
32	1x	2	768	768	no
33	1x	2	640	896	no
34	1x	2	512	1024	no
35	1x	2	384	1152	no
36	1x	2	256	1280	no
37	1x	2	128	1408	no
2 Antennas, 2x					
38	2x	2	640	128	supported
39	2x	2	512	256	supported
40	2x	2	384	384	supported
41	2x	2	256	512	supported
42	2x	2	128	640	supported
3 Antennas, 1x					
43	1x	3	896	128	no
44	1x	3	768	256	no
45	1x	3	640	384	no
46	1x	3	512	512	no
47	1x	3	384	640	no
48	1x	3	256	768	no
49	1x	3	128	896	no
3 Antennas, 2x					
50	2x	3	384	384	supported
51	2x	3	256	512	supported
52	2x	3	128	640	supported
4 Antennas, 1x					
53	1x	4	640	128	no
54	1x	4	512	256	no
55	1x	4	384	384	no
56	1x	4	256	512	no
57	1x	4	128	640	no
4 Antennas, 2x					
58	2x	4	256	128	supported
59	2x	4	128	256	supported
5 Antennas, 1x					
60	1x	5	384	128	no
61	1x	5	256	256	no
62	1x	5	128	384	no
5 Antennas, 2x					
63	2x	5	128	128	supported
6 Antennas, 1x					
64	1x	6	384	128	no
65	1x	6	256	384	no

Mode	Rate	# Ant	# Taps	# Hypothesis	OQPSK*
66	1x	6	128	512	no
6 Antennas, 2x					
67	2x	6	128	128	supported
7 Antennas, 1x					
68	1x	7	256	128	no
69	1x	7	128	256	no
8 Antennas, 1x					
70	1x	8	258	128	no
71	1x	8	128	256	no

14.5 RTOS Interface Examples

This section gives examples of how to interface to the VxWorks RTOS (refer to Section 2.3.3).

14.5.1 VxWorks RTOS Interface Example

The following is an example of how to create an RTOS interface, using VxWorks, to the CBME.

14.5.1.1 m_rtos.h

This file is provided by Morphics. Some parts may be modified, and these are identified by comments.

```
#ifndef __M_RTOS_H
#define __M_RTOS_H

/*----------------*/
/* queue messages */
/*----------------*/
/* Error Queue */
#define   SEARCHER_QUEUE_OVERFLOW_MSG            0x0000
#define   PDE_QUEUE_OVERFLOW_MSG                 0x0001
#define   COMBINER_DSP_QUEUE_OVERFLOW_MSG        0x0002
/* PDE Queue */
#define   PDE_ENERGY_MSG                         0x0100
/* Searcher Queue */
#define   SEARCHER_ENERGY_MSG                    0x0200
/* Combiner DSP Queue */
#define   COMBINER_DSP_MSG                       0x0300
#define   FINGER_OFFSET_MSG                      0x0301

/* max messages per queue; these values may be increased to prevent */
/* queue overflow                                                   */
#define    PDE_QUEUE_MAX_MSG_COUNT               20
#define    SEARCHER_QUEUE_MAX_MSG_COUNT          20
#define    COMBINER_DSP_QUEUE_MAX_MSG_COUNT      20
#define    ERROR_QUEUE_MSG_COUNT                 10


/* max message size (in bytes) per queue; do not change these values */
#define    PDE_QUEUE_MAX_MSG_SIZE               144
#define    SEARCHER_QUEUE_MAX_MSG_SIZE           28
#define    COMBINER_DSP_QUEUE_MAX_MSG_SIZE       16
#define    ERROR_QUEUE_MAX_MSG_SIZE               4

/*-------------------------------------------------------------------
*/
/* list of queues that VMI will create, along with maximum queue index
*/
/* used for error checking
*/
/*-------------------------------------------------------------------
*/
typedef enum
```



```
{
  SEARCHER_Q,
  PDE_Q,
  COMBINER_Q,
  ERROR_Q,
  MAX_Q_ENUM

} VMI_MSG_Q_ENUM;
```



```
/*-------------------------*/
/* array of message queues */
/*-------------------------*/
extern MSG_Q_ID q_list[MAX_Q_ENUM];


/*---------------------*/
/* function prototypes */
/*---------------------*/
void * VMI_Msg_Queue_Create( VMI_MSG_Q_ENUM q_type,
                            UINT16          max_msg_length,
                            UINT16          max_msgs);


UINT16  VMI_Msg_Queue_Send(
       VMI_MSG_Q_ENUM q_type,
       UINT32  *p_msg,
       UINT16  msg_length);

#endif /* __M_RTOS_H */
```


14.5.1.2 m_rtos.c

This file contains code that is RTOS-specific.

```
#include "vxworks.h"
#include "m_rtos.h"

/* global declaration for array of message queues */
MSG_Q_ID q_list[MAX_Q_ENUM];

void * VMI_Msg_Queue_Create( VMI_MSG_Q_ENUM q_type,
                             UINT16  max_msg_length,
                             UINT16  max_msgs)

{
   int status = M_SUCCESS;

   /* create the message queue */
   q_list[q_type] = msgQCreate(max_msgs, max_msg_length, MSG_Q_FIFO);

   /* if queue creation failed */
   if (!q_list[q_type])
   {
     status = M_RTOS_MSG_QUEUE_CREATE_ERROR;
   }

  return status;
}


UINT16  VMI_Msg_Queue_Send(VMI_MSG_Q_ENUM q_type,
                           UINT32  p_msg,
                           UINT16  msg_length
                          )
{
   int status;

   /* send message to designated message queue */
   status = msgQSend(q_list[q_type],
                     msg,
                     msg_length,
                     NO_WAIT,
                     MSG_PRI_NORMAL);
   if (stat == OK)
   {
      return (M_SUCCESS);
   }
   else
   {
```



```
        return (M_RTOS_MSG_QUEUE_SEND_ERROR);
    }
}
```


14.6 Acronyms

CBME	Cellular Basestation Modem Engine
CGU	Code Generation Unit
DLL	Delay-Lock Loop (to track chip timing)
DSM	Dwell State Machine (associated with searcher)
FLL	Frequency Lock Loop (to track frequency offsets due to relative motion of mobile)
LFSR	Linear Feedback Shift Register
PDE	Preamble Detection Engine
PDP	Power Delay Profile
RTOS	Real Time Operating System
VMI	Virtual Machine Interface; specifically, the Morphics CBME VMI.

Morphics Technology

Morphics Technology is a communications systems design company specializing in reconfigurable digital signal processing architectures and algorithms. Morphics supplies ICs and licenses IP cores that bring unprecedented computational efficiency and provide flexibility and scalability to wireless communications systems. Morphics IC and IP product platforms support a variety of applications such as cellular, fixed wireless, unlicensed wireless LAN's, cordless telephony, personal basestations and telemetry.

Document number: VMI-CDMA-BTS-KH2.01

For additional information or product support please contact:
Morphics Technology, Inc., 675 Campbell Technology Parkway, Suite 100, Campbell, CA 95008-5059 USA

Telephone: +1.408.369.7227 • FAX: +1.408.369.7210 • E-mail: info@morphics.com • http://www.morphics.com